Loon

Interactive graphics in R

R.W. Oldford

UNIVERSITY OF WATERLOO

The loon package

Loon is an interactive visualization system built using tcltk.

The loon package is available on CRAN. To install the package start your R and run

```
install.packages('loon')
```

You can also install the latest development release (make sure it looks like it is passing the "builds") directly from GitHub with the following R code (you might need to install devtools)

```
devtools::install_github("great-northern-diver/loon", subdir = "R")
```

Once installed, the loon package is loaded in the usual way:

```
library(loon)
```

And instructions on loon are available in different ways:

```
l_help()                    # loon's web overview
l_web()                     # loon's web manual
help(package = "loon")      # loon's R help pages
```

l_plot()

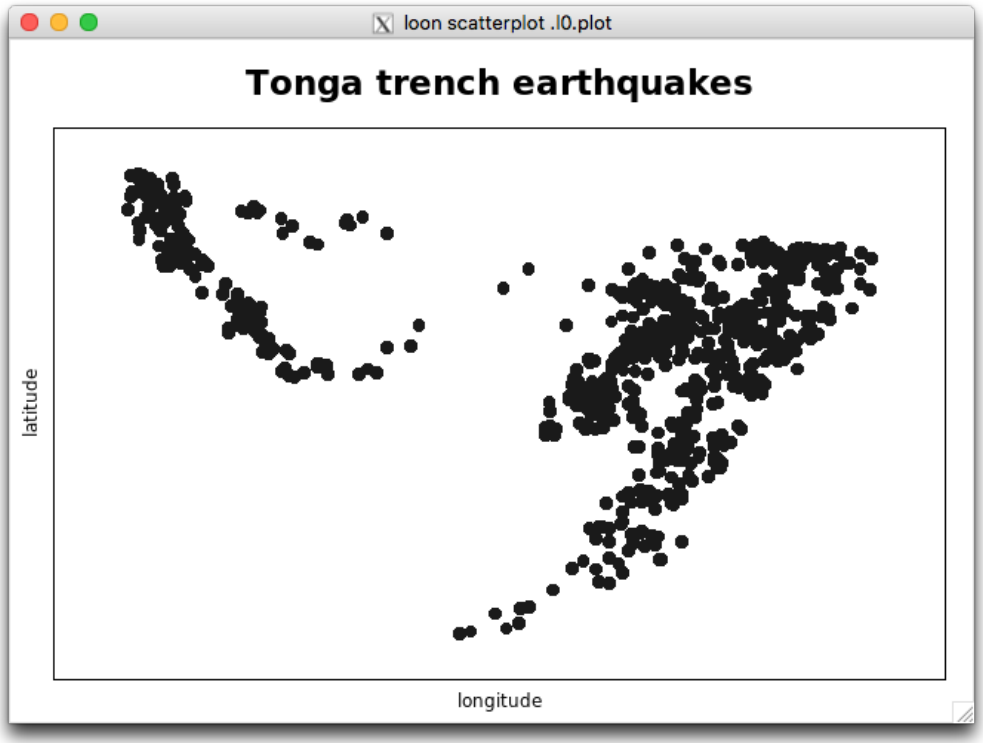
The basic plot function in `loon` is similar to that of `plot()` from the base `graphics` package.

```
# Tonga Trench Earthquakes
p <- l_plot(quakes$long, quakes$lat, title = "Tonga trench earthquakes",
            xlabel = "longitude", ylabel ="latitude")
# produces the scatterplot
```

l_plot()

The basic plot function in `loon` is similar to that of `plot()` from the base `graphics` package.

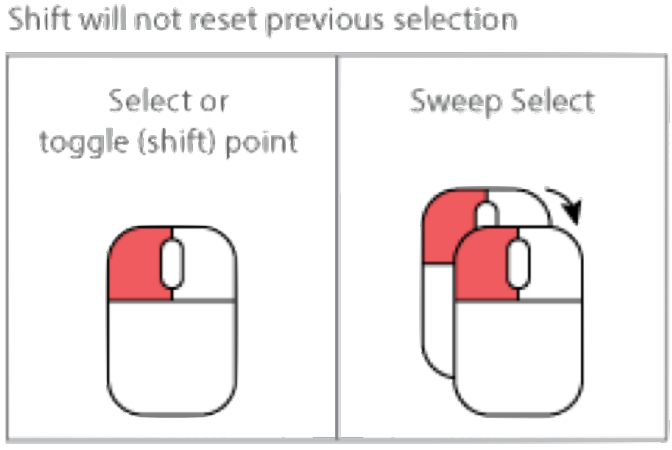
```
# Tonga Trench Earthquakes
p <- l_plot(quakes$long, quakes$lat, title = "Tonga trench earthquakes",
            xlabel = "longitude", ylabel ="latitude")
# produces the scatterplot
```



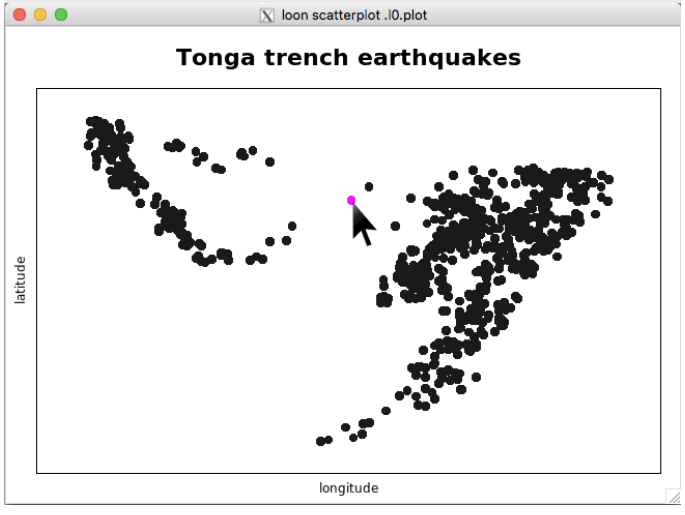
Interact with the plot using the mouse and keyboard.

Mouse gestures - selection

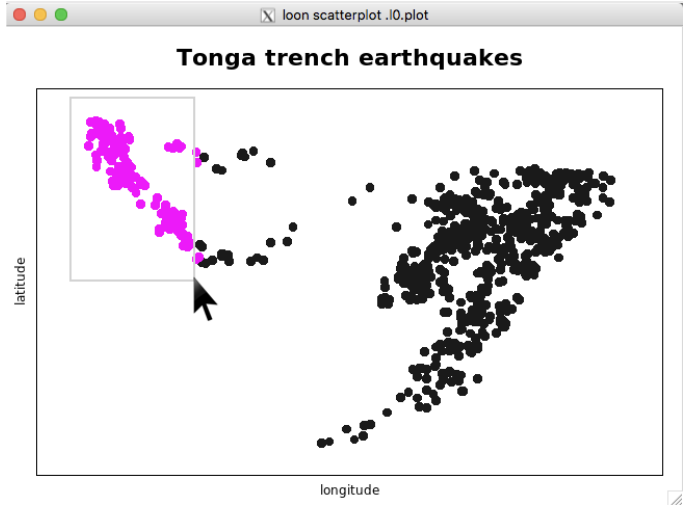
Points can be selected



One at a time

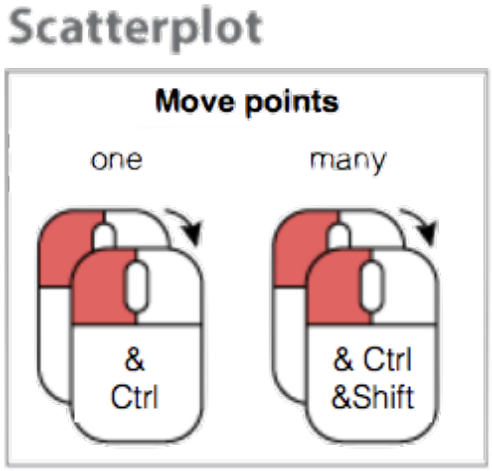


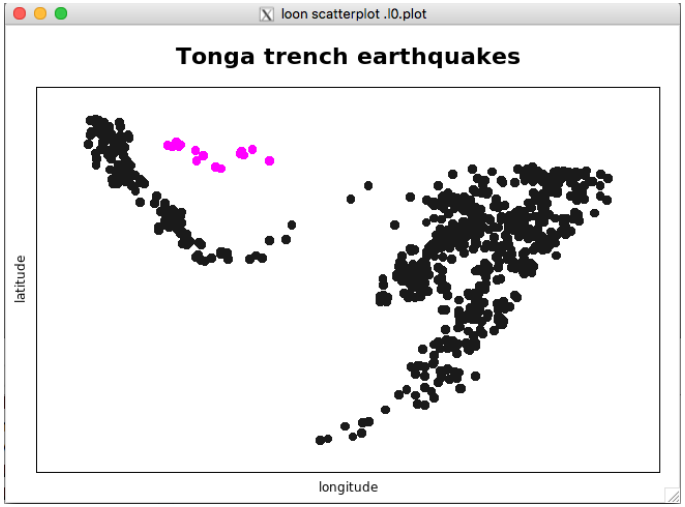
or by "sweeping" out a rectangle

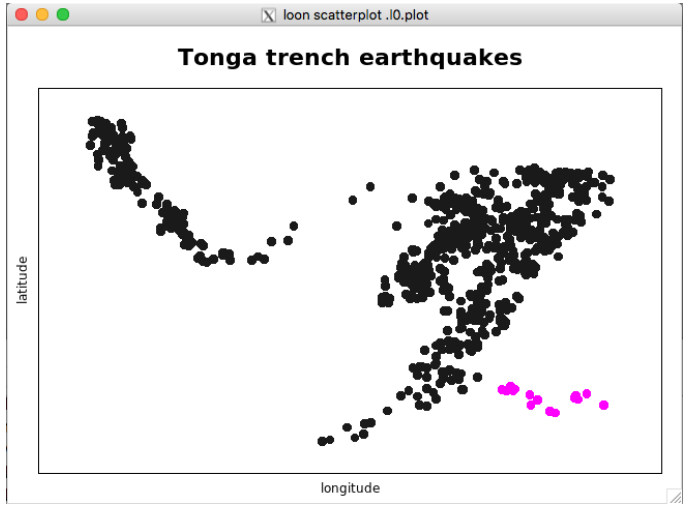


Mouse gestures - moving points

Selected points can be moved

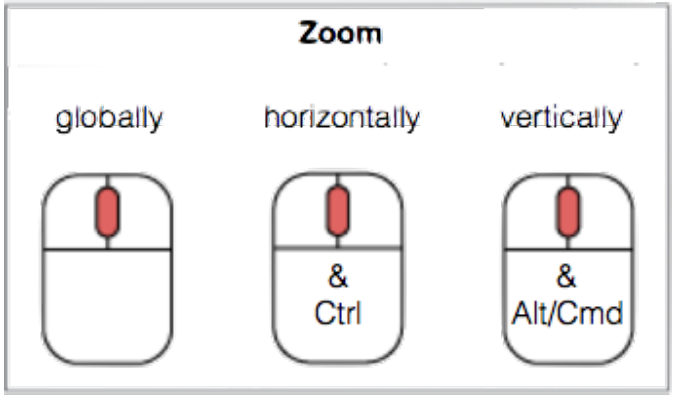




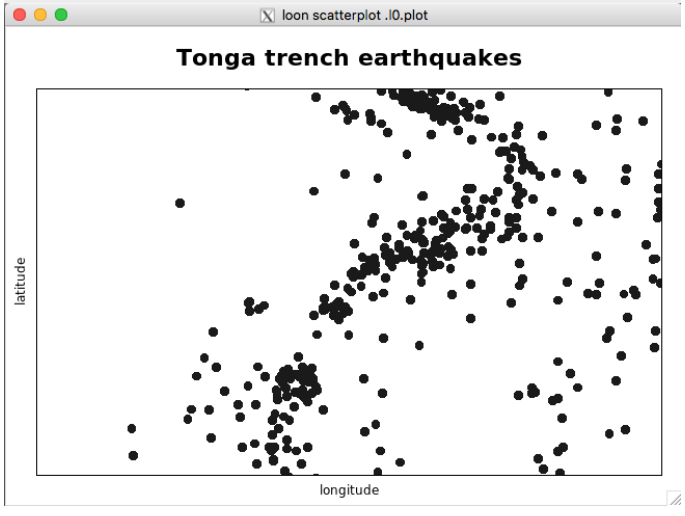


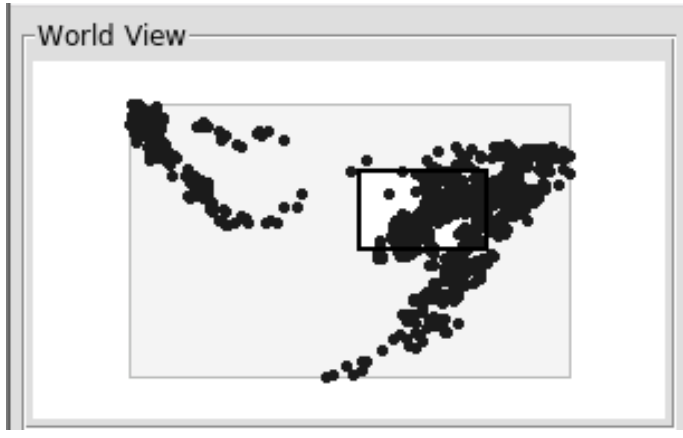
Mouse gestures - zooming

Zooming (on plot OR on "World View")



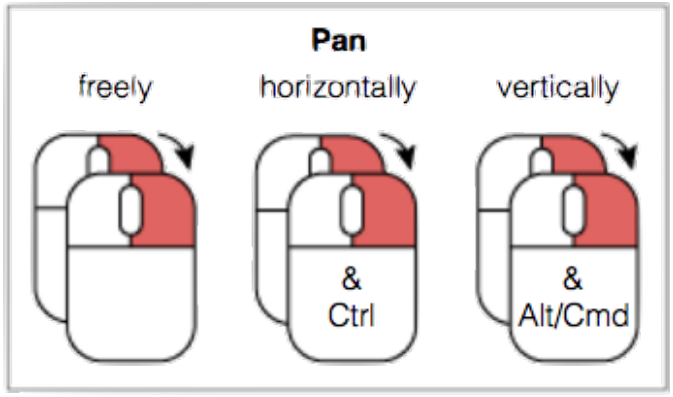
Zooming (globally)



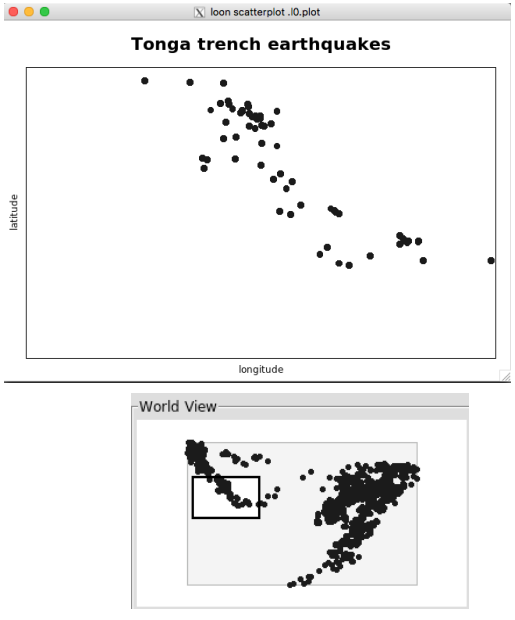


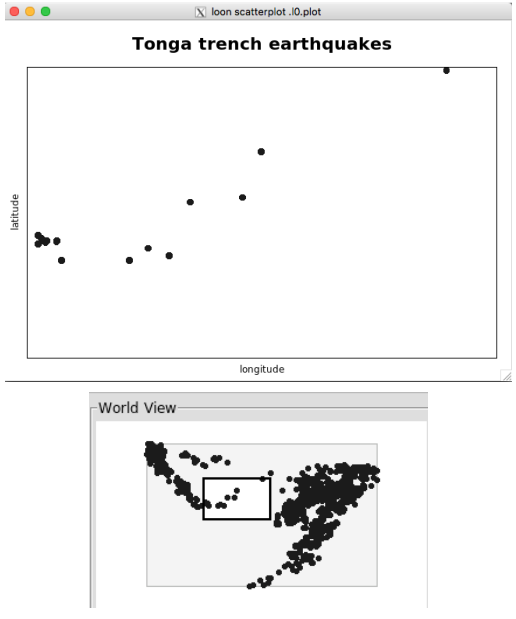
Mouse gestures - panning

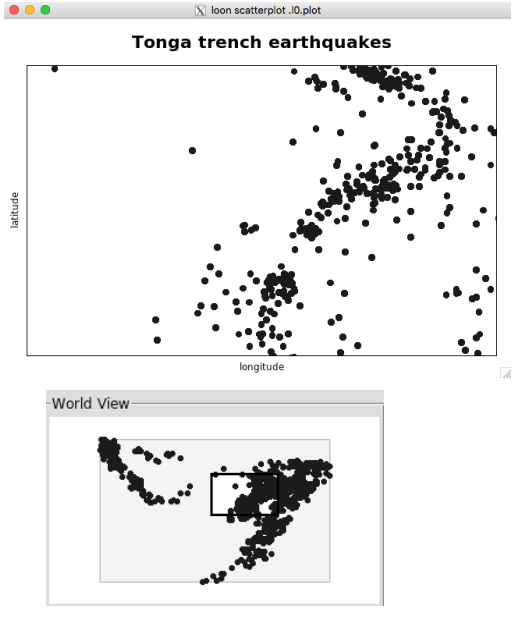
Panning (on plot OR on "World View")



Panning (horizontally)

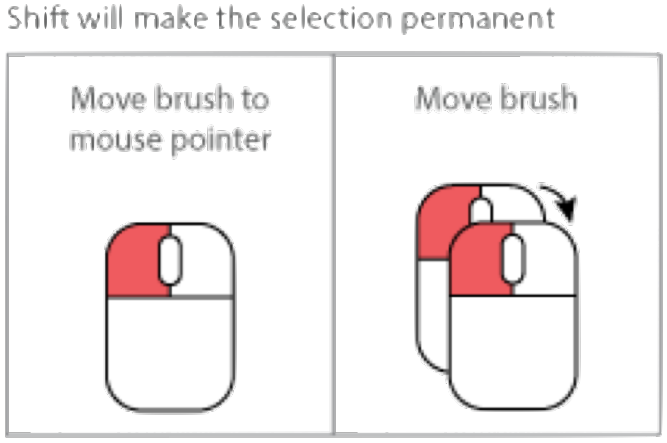


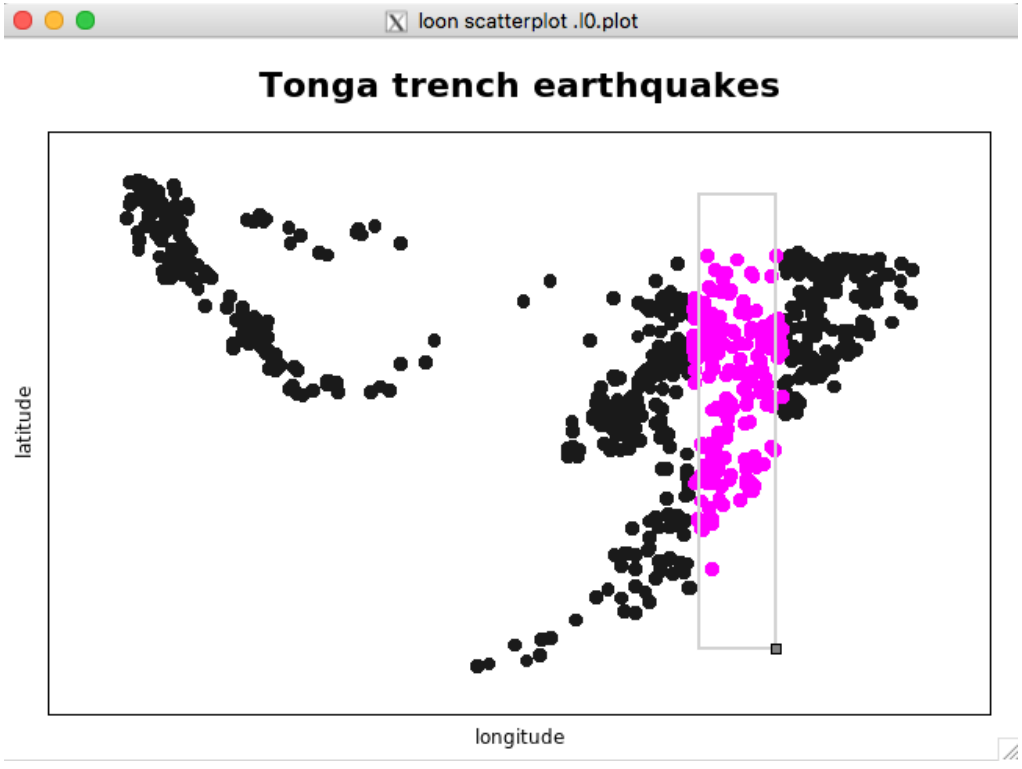




Mouse gestures - brushing

Brushing (selection via a fixed size rectangle)



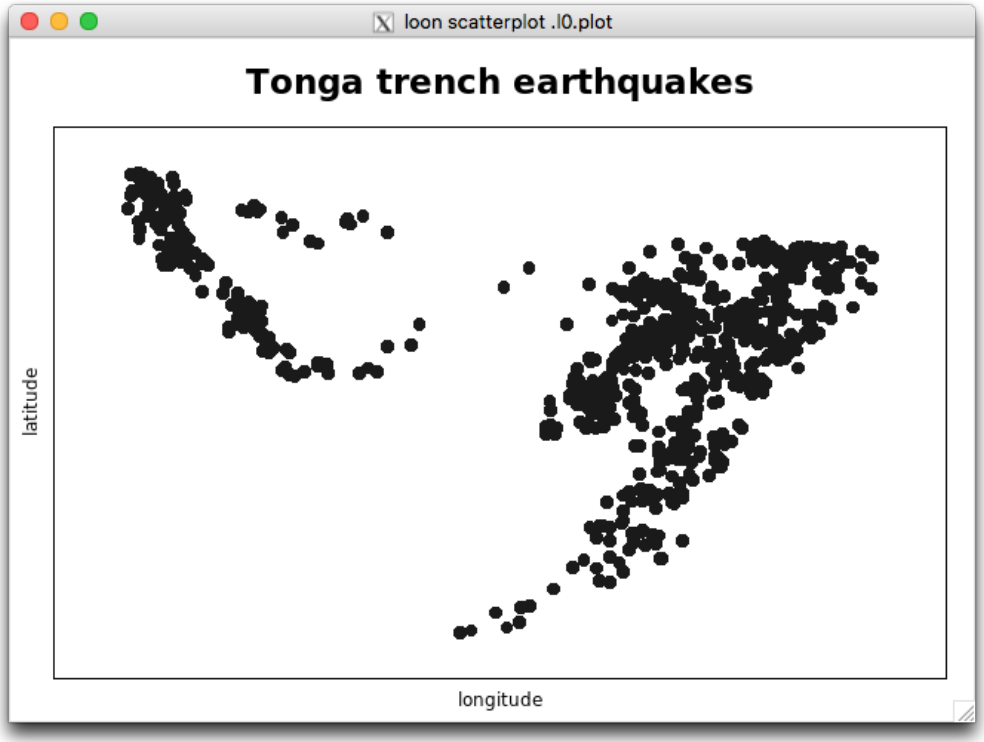


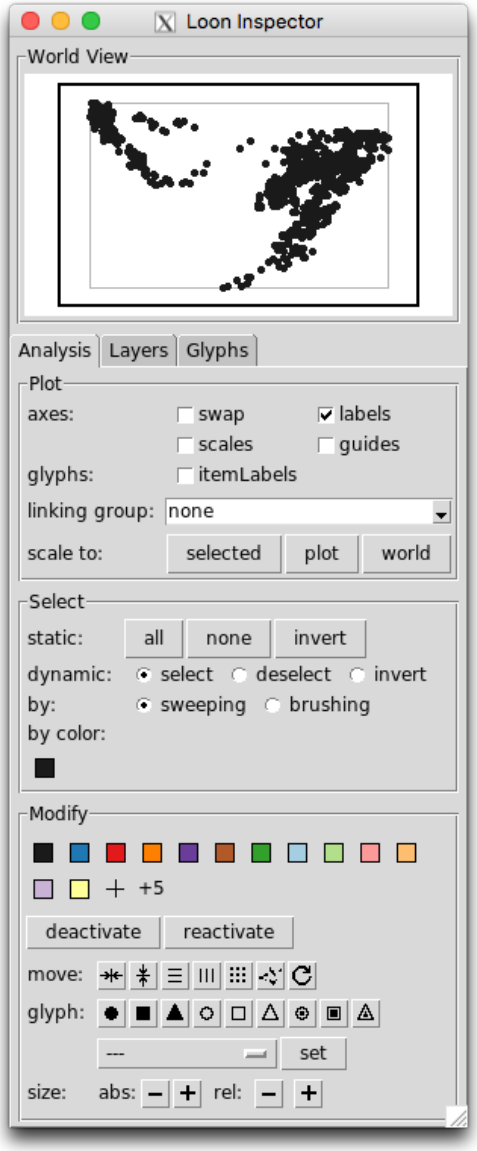
The rectangular "brush"

- maintains its shape as it is moved
- can be reshaped by selecting the small grey square
- tall thin is equivalent to selecting x values
- wide flat to selecting ys

The loon inspector

Whenever a loon plot is active, the **loon inspector** is focussed on that plot:

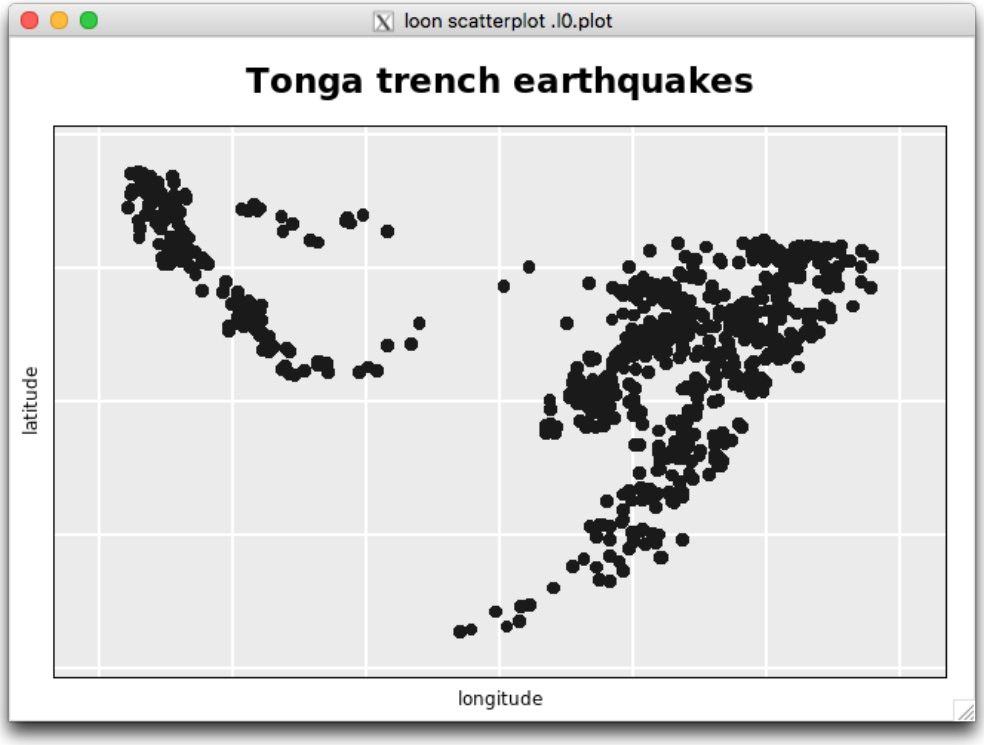


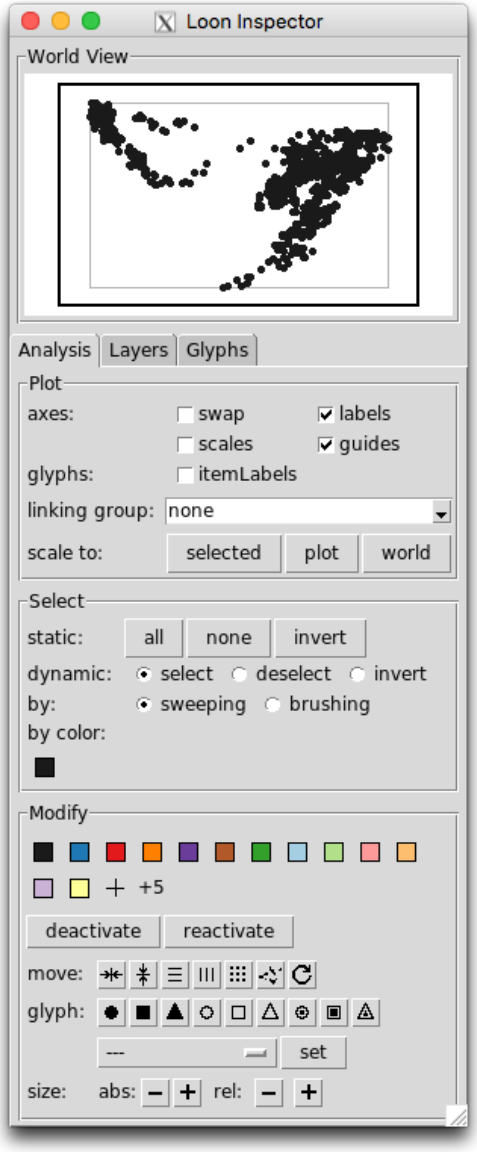


The loon inspector is created when the first loon plot is and it can never be closed while there are any loon plots.

The loon inspector - interacting with the plot

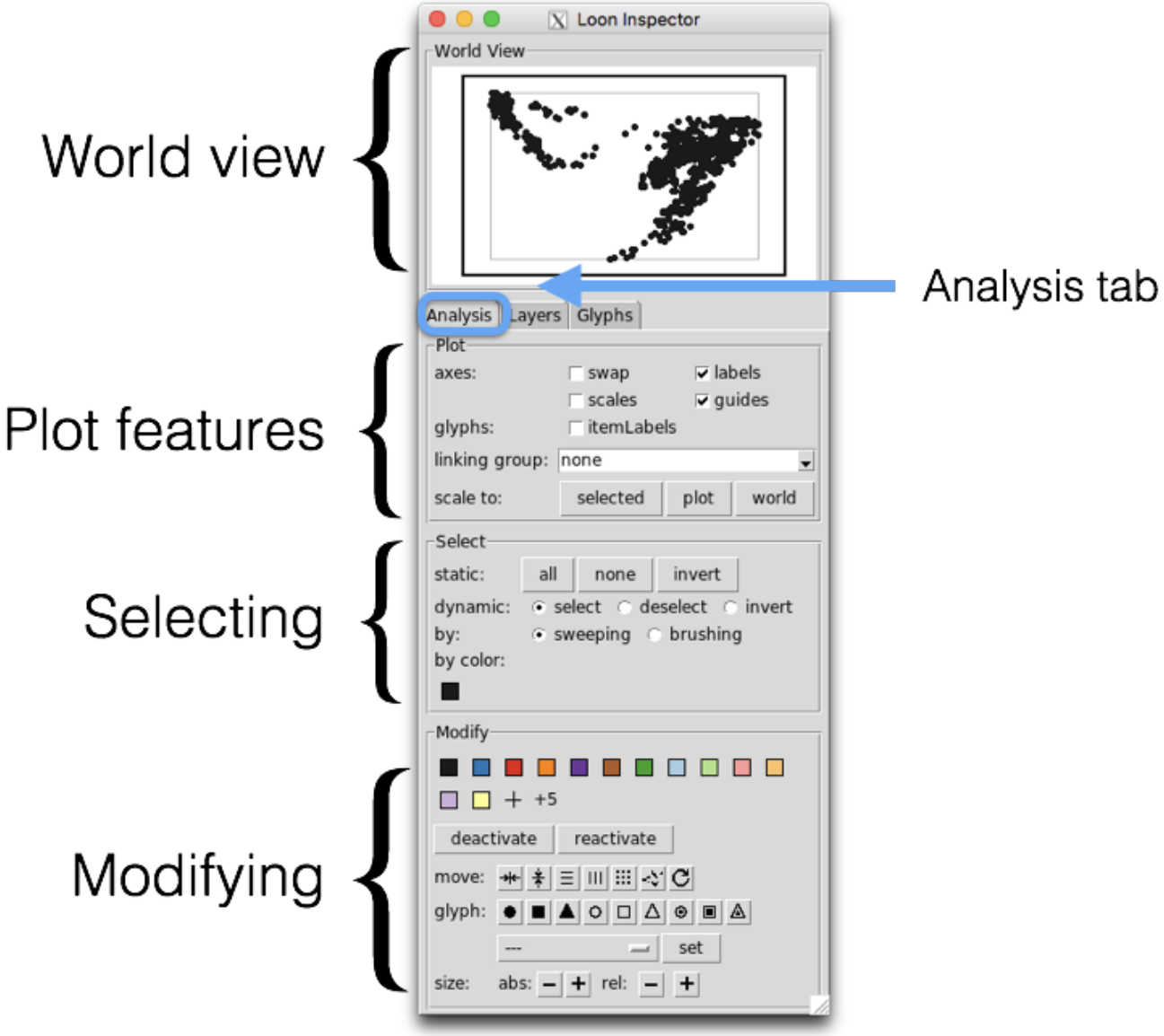
The inspector is used to change the loon plot. For example, checking the "guides" box places a background grid of guide lines on the loon plot.



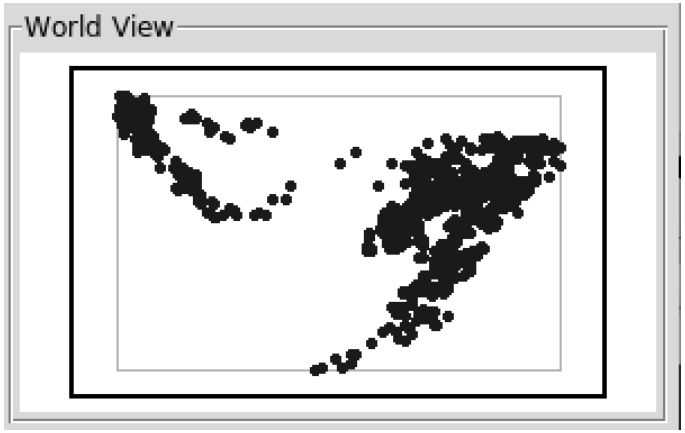


The loon inspector - components

The loon inspector separates into different panels on the analysis tab:



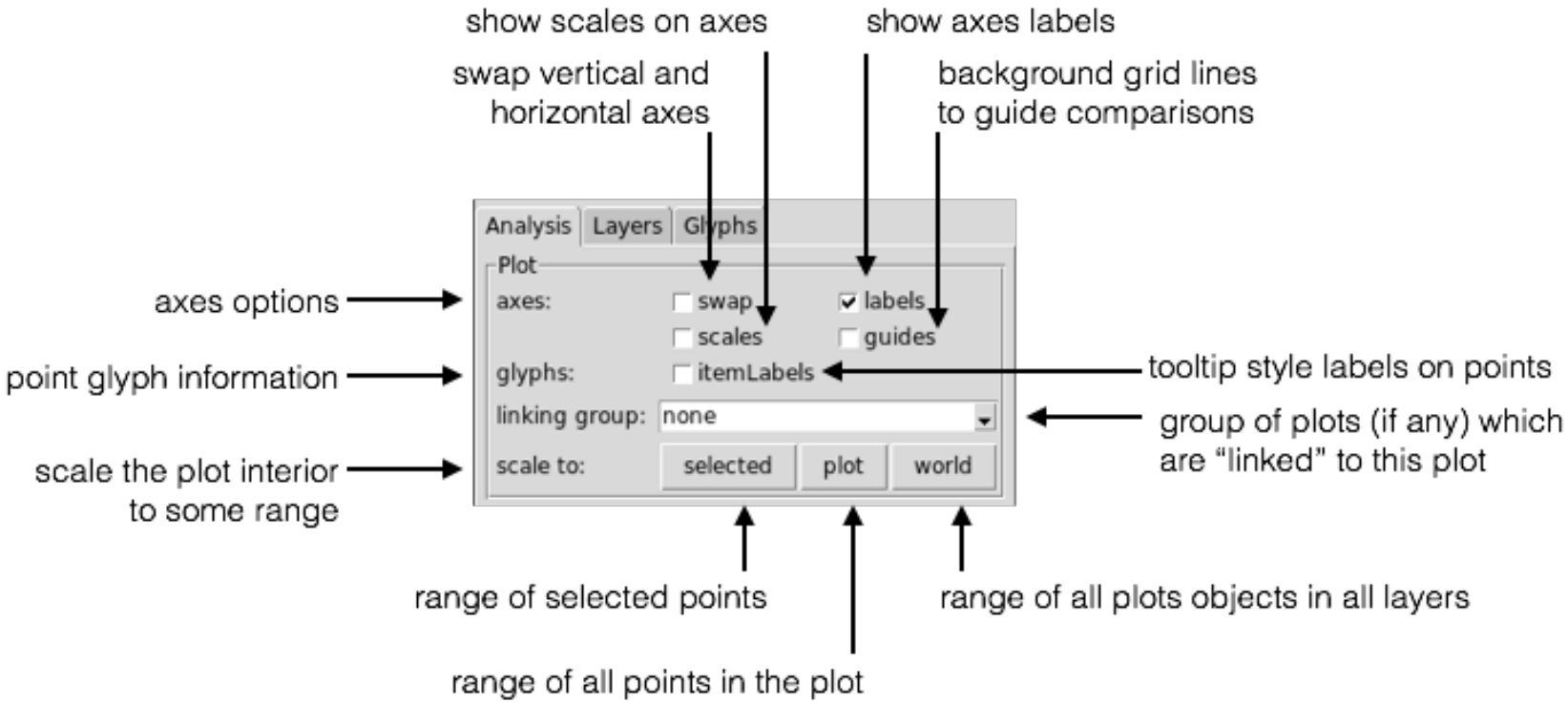
The loon inspector - the World View



The world view always shows the **whole** of the current plot, what is displayed and what is active.

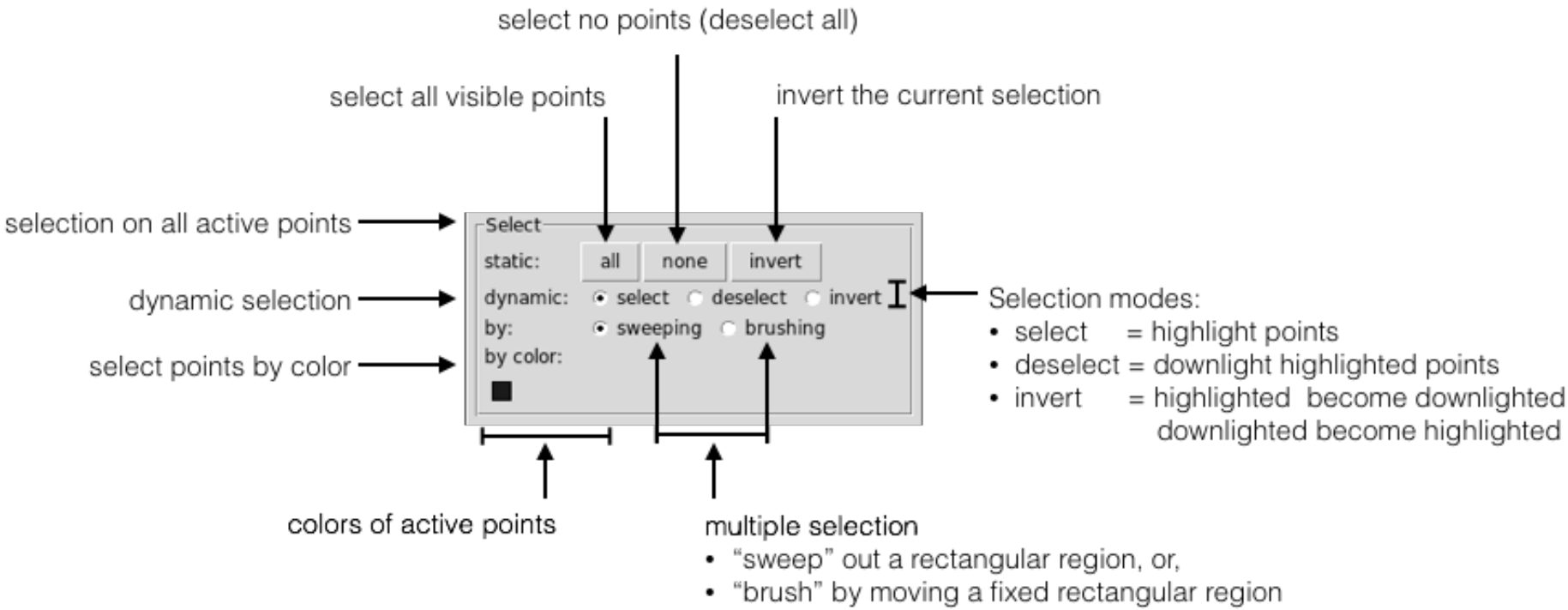
The loon inspector - the Analysis tab

Plot features:



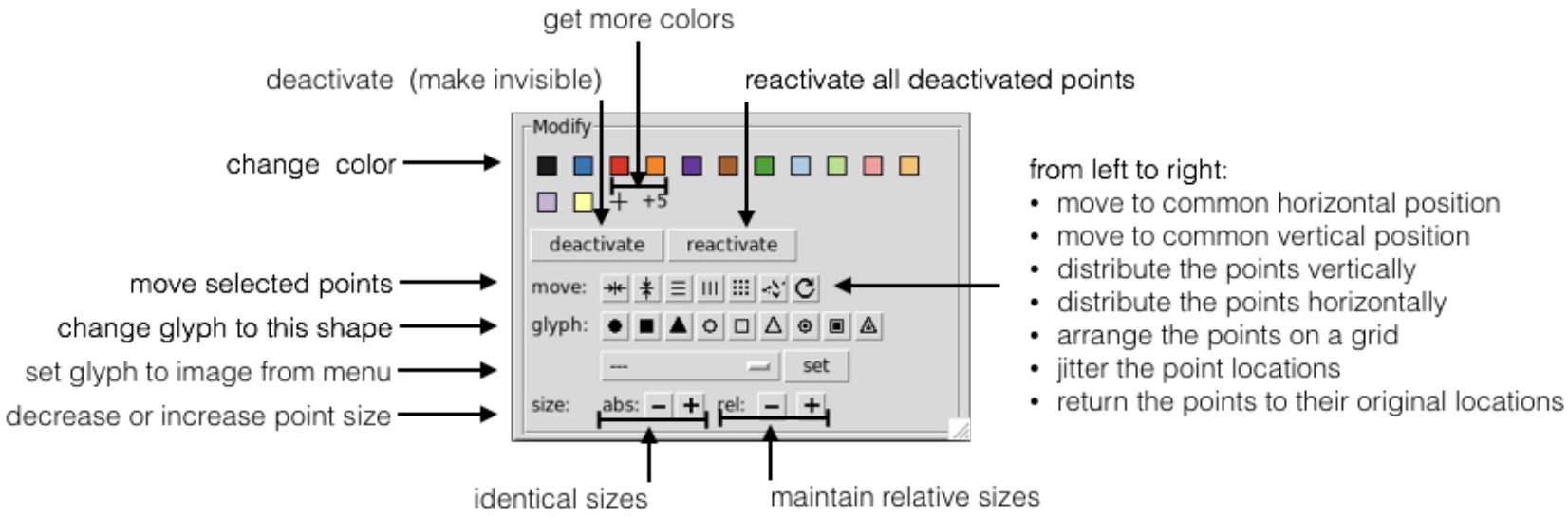
The loon inspector - the Analysis tab

Point selection:



The loon inspector - the Analysis tab

Modifying selected points:

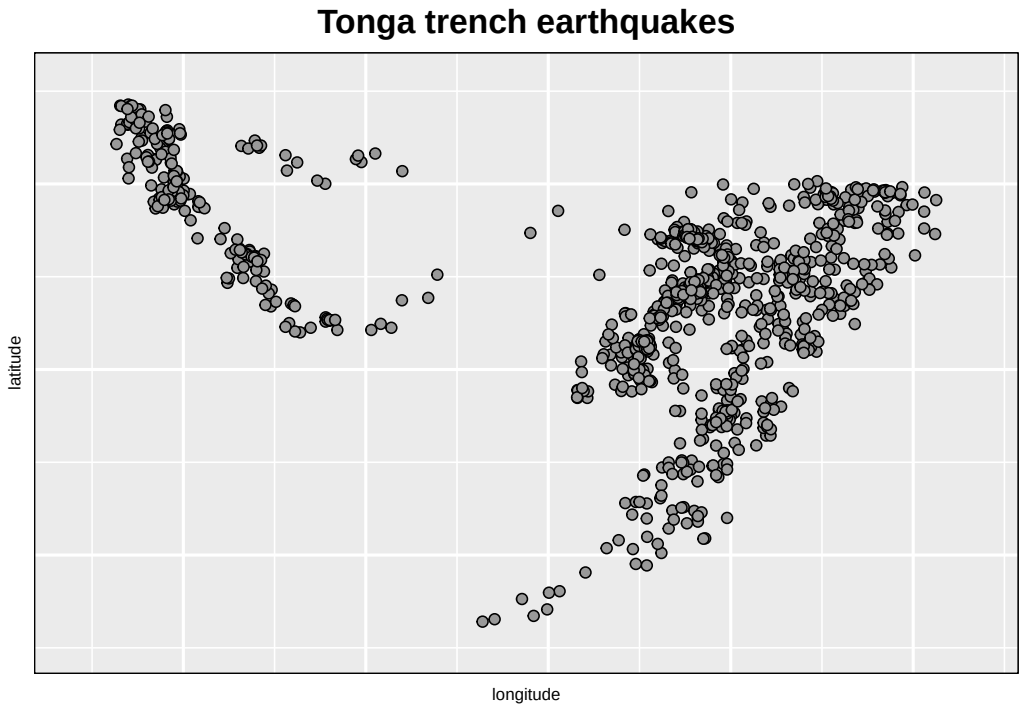


Printing the plot

At any time the **current view** of the loon plot `p` can be transferred to a static `grid` graphics plot simply as `plot(p)` and from there saved as usual. (More on this later.)

For example,

```
plot(p)
```



Adding layers

The plot is a data structure (in `tcltk`) and we can add other plot objects, such as polygons (and other geometric structures), to it.

For example, we can add a map to the current loon plot `p`.

First get the relevant map:

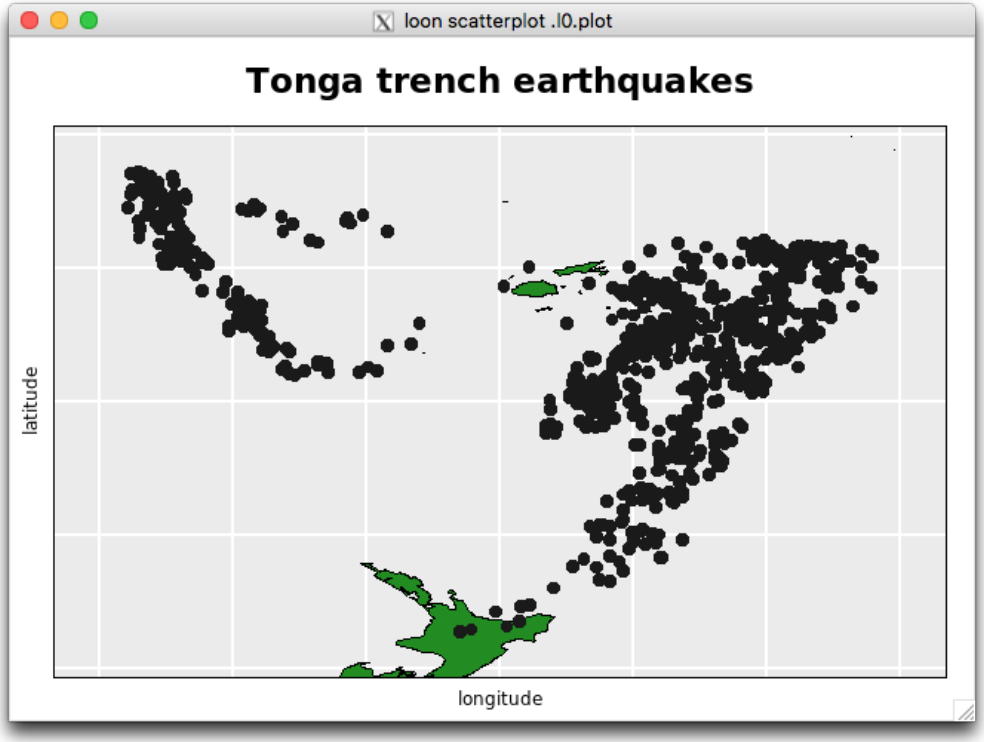
```
library(maps)
NZFijiMap <- map("world2", regions=c("New Zealand", "Fiji"), plot=FALSE)
```

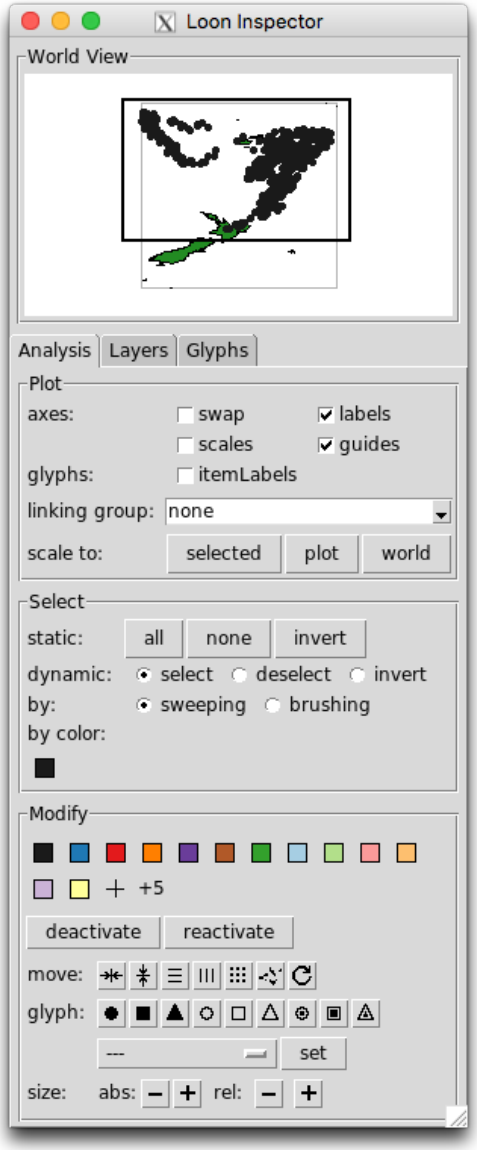
It is added as a "layer" to the loon plot

```
l_layer(p, NZFijiMap,
        label = "New Zealand and Fiji",
        color = "forestgreen",
        index="end")
```

```
## loon layer "New Zealand and Fiji" of type polygons of plot .l0.plot
## [1] "layer0"
```

Adding layers - maps



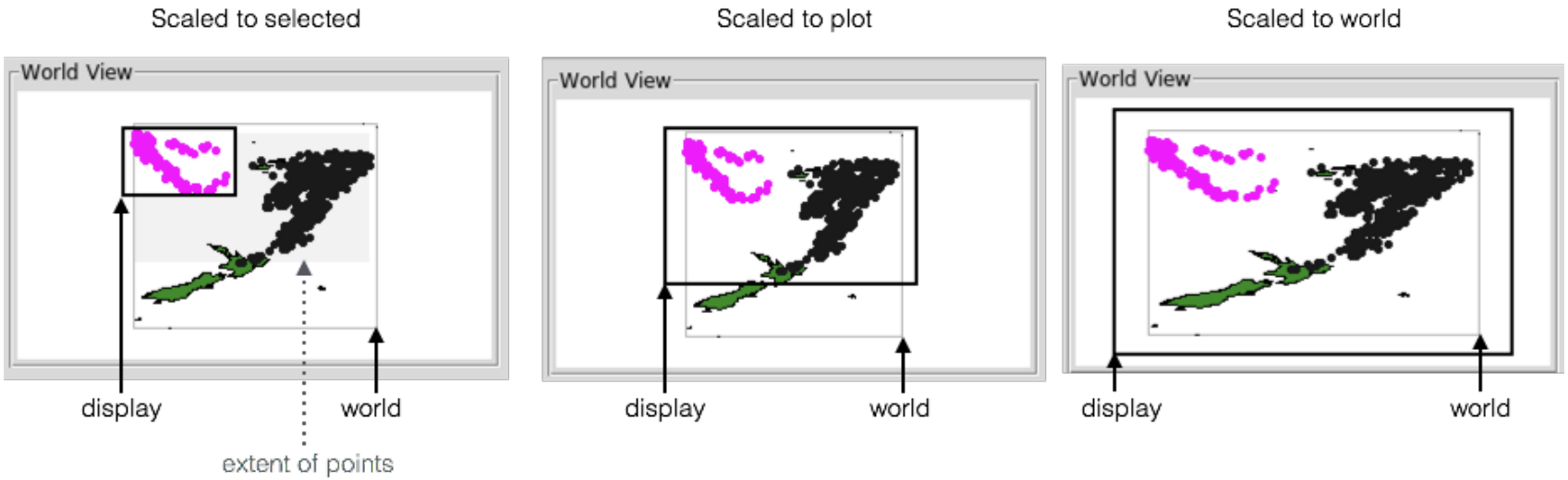


As can be seen in the world view, much of the map is outside of the plot display (shown as the black-bordered rectangle in the world view).

Analysis tab - scaling choices

Adding the map allows us to see the effect of the three plot scaling choices which are available from the inspector.

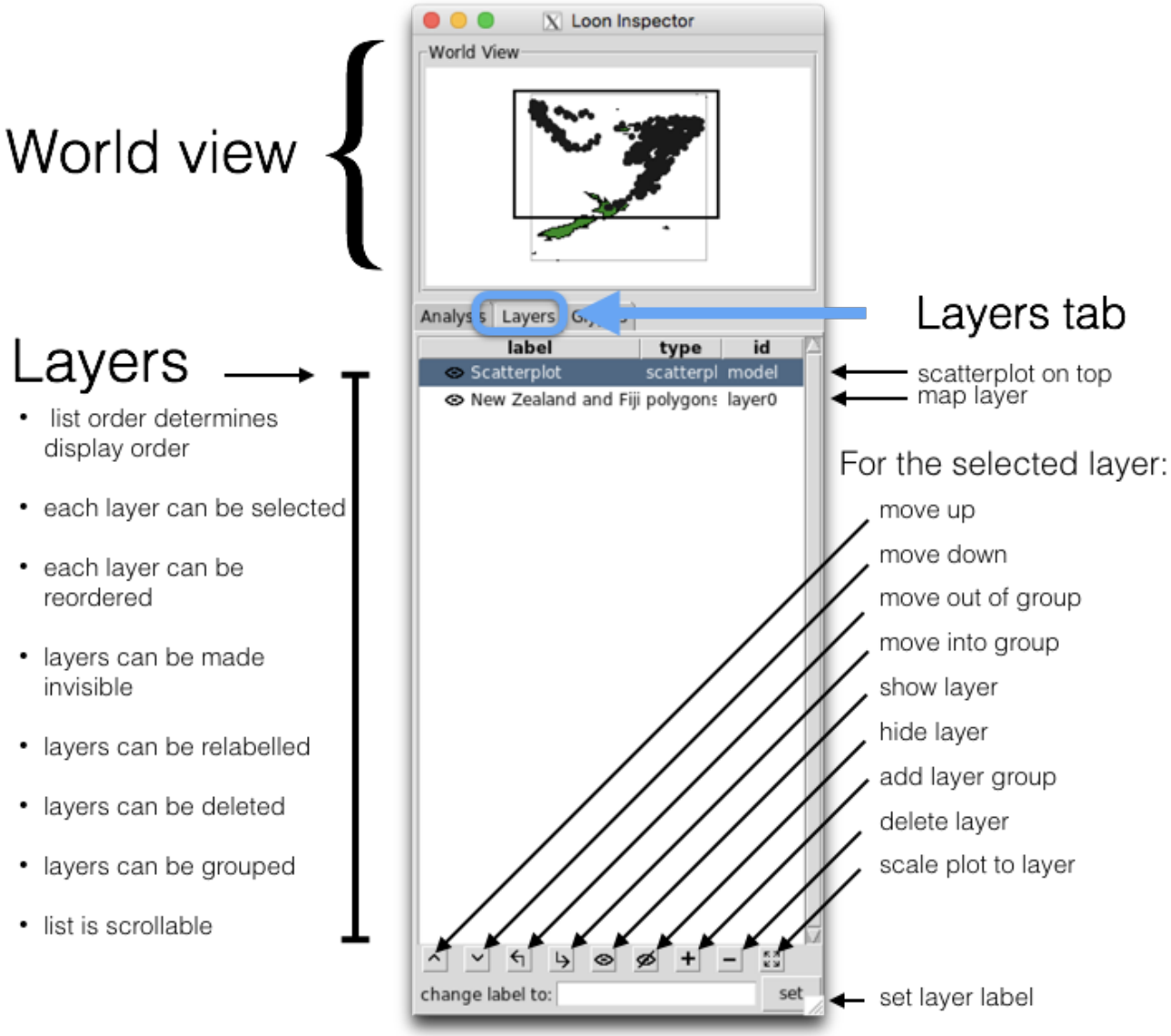
The effect is best seen on the world view:



Note that the plot in the world view matches that of the actual plot. Also the aspect ratio changes with the scaling.

Loon inspector - Layers tab

The map is added as a layer, which can be seen by selecting the "Layers" tab in the inspector:



More than one plot - linking

The quakes data actually contains measurements on several variates:

```
str(quakes)
```

```
## 'data.frame':    1000 obs. of  5 variables:
##  $ lat     : num  -20.4 -20.6 -26 -18 -20.4 ...
##  $ long    : num  182 181 184 182 182 ...
##  $ depth   : int  562 650 42 626 649 195 82 194 211 622 ...
##  $ mag     : num  4.8 4.2 5.4 4.1 4 4 4.8 4.4 4.7 4.3 ...
##  $ stations: int  41 15 43 19 11 12 43 15 35 19 ...
```

We might construct a second plot of the quake magnitude versus its depth:

```
p2 <- l_plot(quakes[,c("depth", "mag")],  ylabel = "magnitude",
             showScales = TRUE, showGuides = TRUE,
             linkingGroup = "quakes")
```

Notes:

- the data are given here as a data frame of two variates
- we specified that both ‘guides‘ and ‘scales‘ be on
- ‘p2‘ is assigned the string "quakes" as its "linkingGroup"

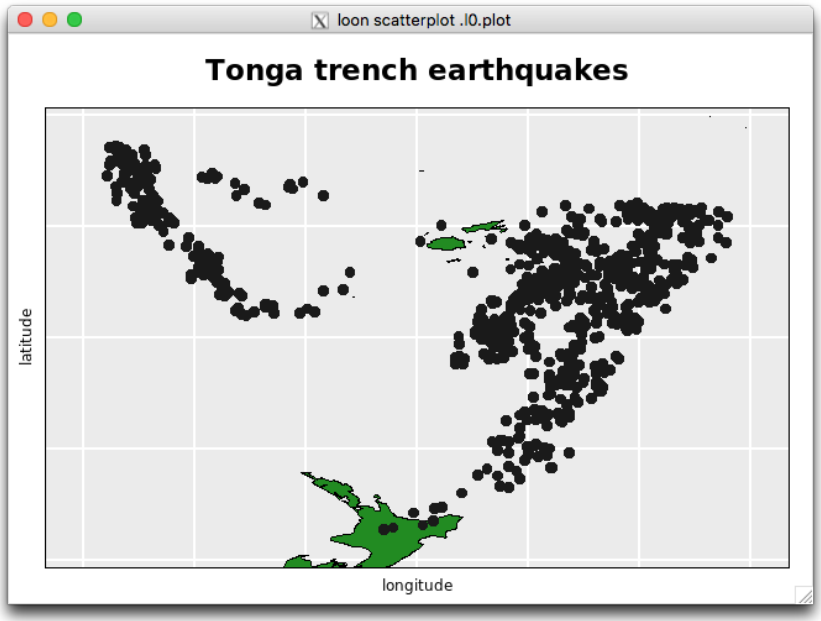
We can ensure that the original p participates in the same linking group by either selecting “quakes” from the inspector for p or by setting p’s linking group directly:

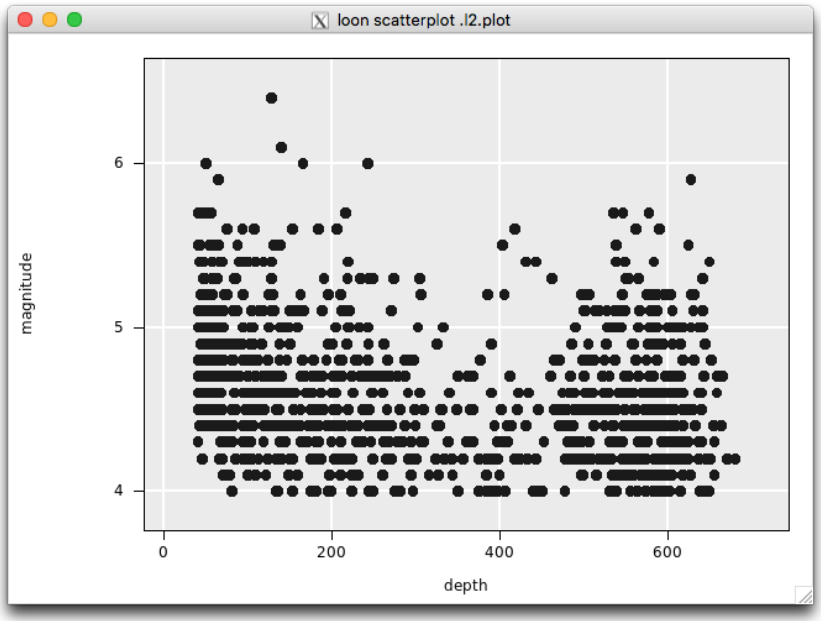
```
l_configure(p, linkingGroup = "quakes", sync="pull")
```

This required a value for sync (i.e. synchronize) which here tells p to “pull” its values from the linking group.

More than one plot - linking

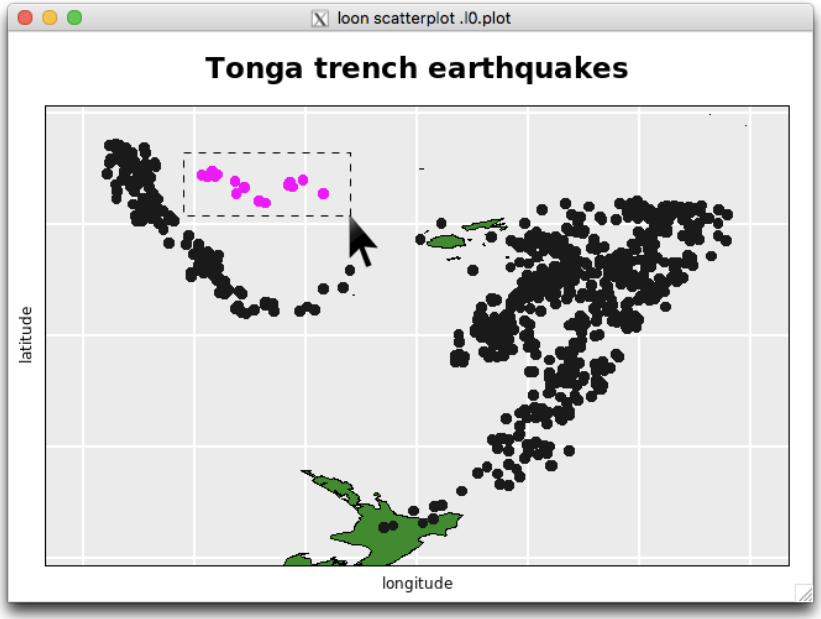
The second plot (p2) shows the quake magnitude versus its depth.

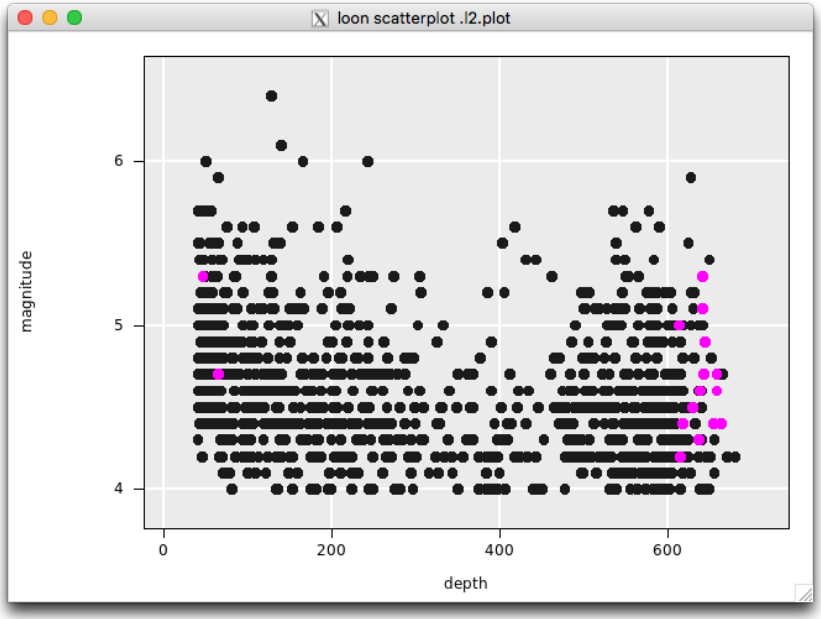




Linking - selection queries

Because the two plots are linked (in the same `linkingGroup`) we can form queries on one plot by using selection (here `sweep`) and see the results (highlighted points) on the second plot:

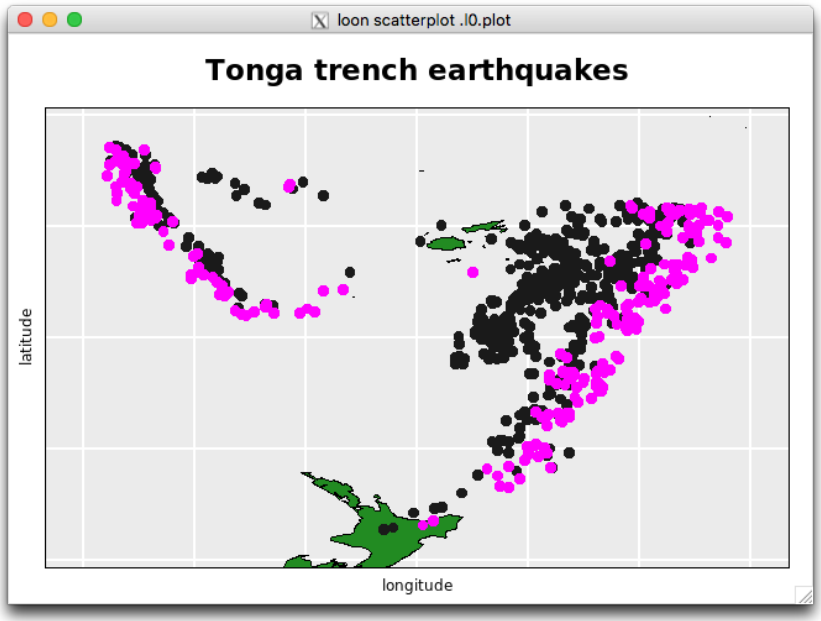


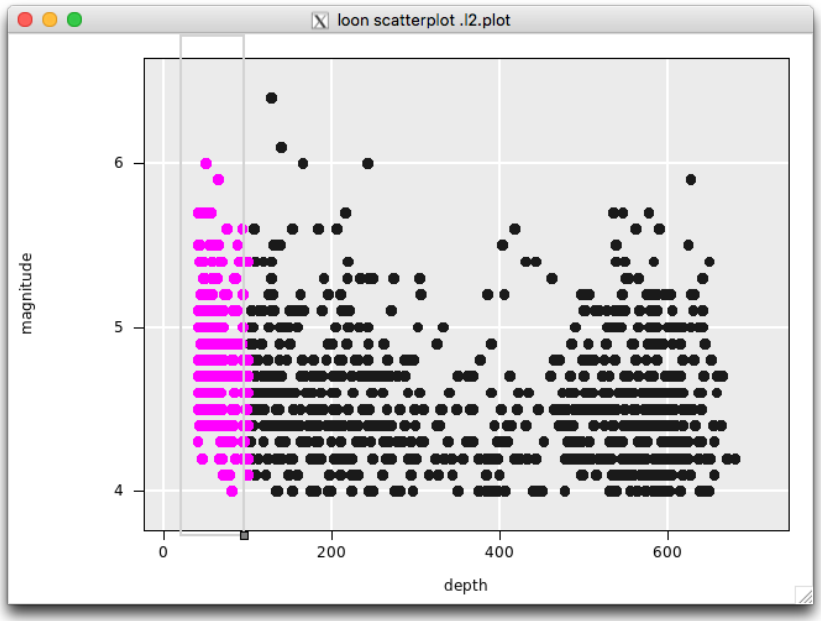


Earthquakes at this location seem to be mostly very deep, with only a couple of relatively shallow quakes. The magnitude of earthquakes in this region seem to relatively spread out (none are amongst the greatest or least magnitude quakes).

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

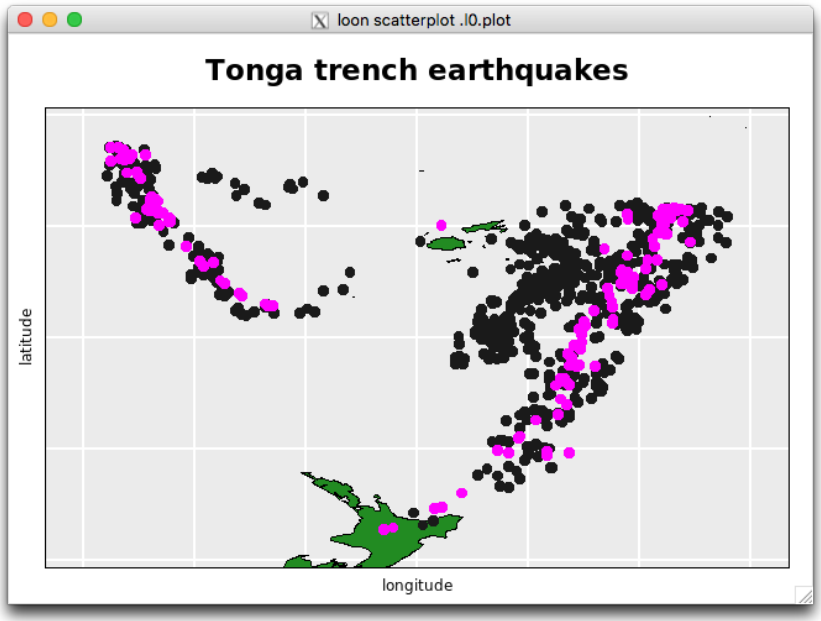


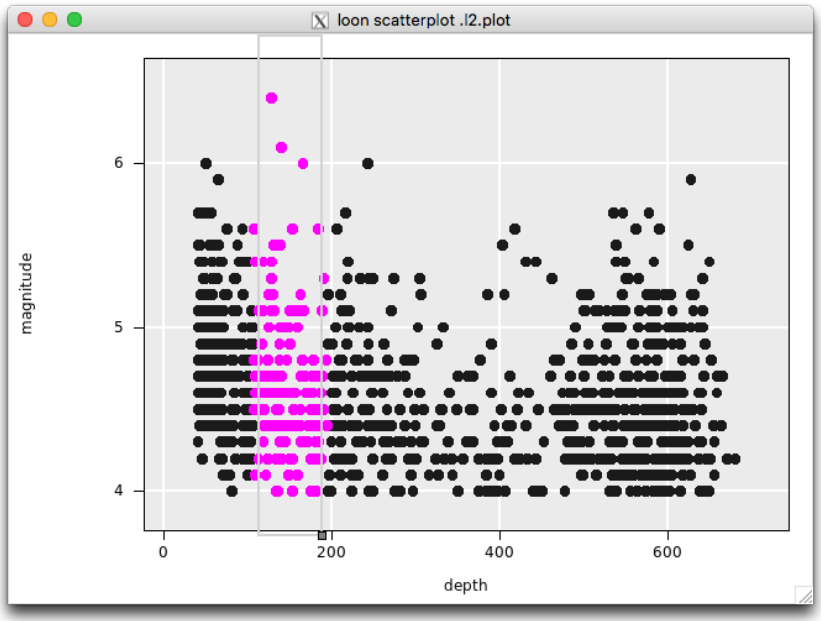


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

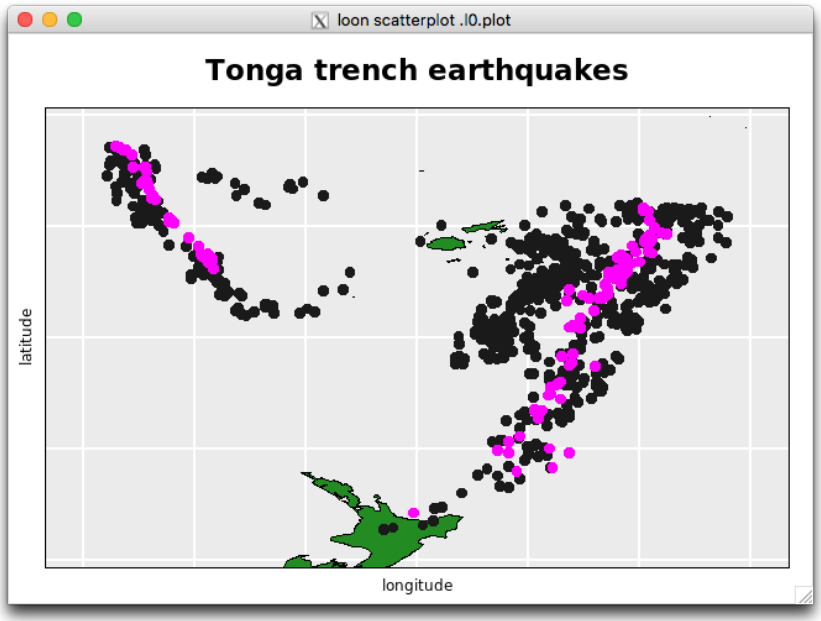


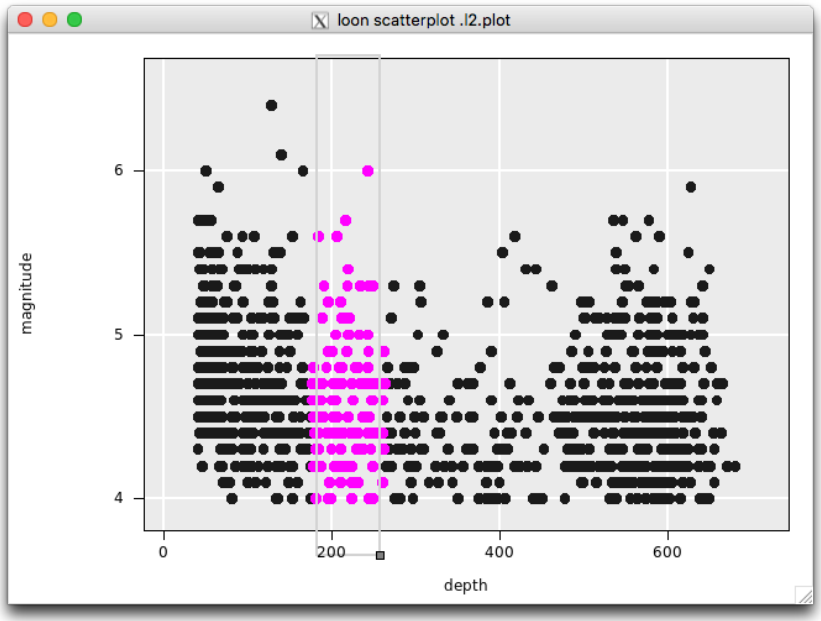


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

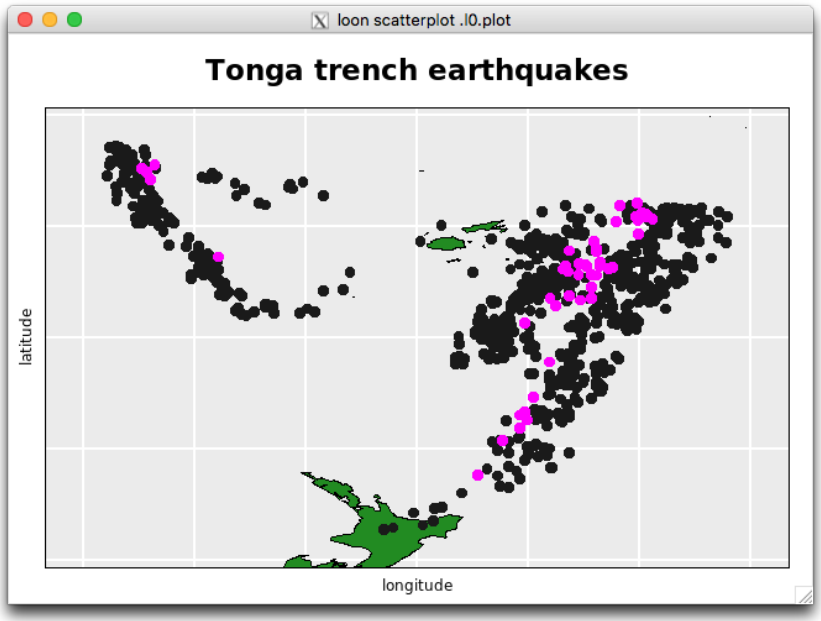


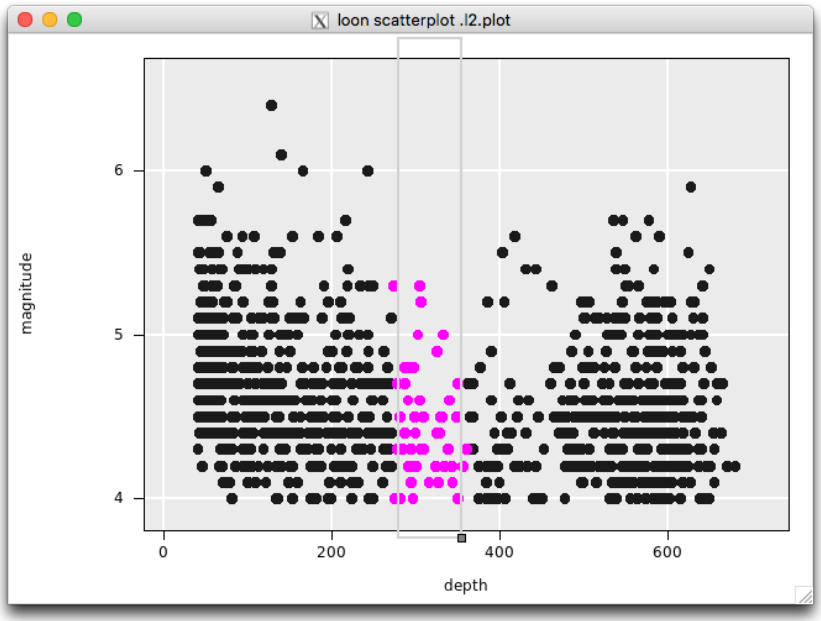


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushin

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

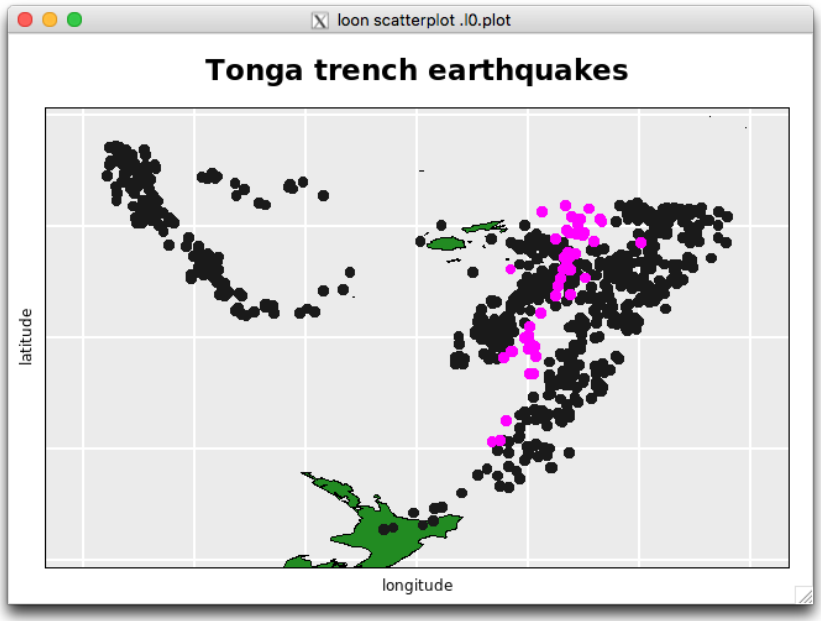


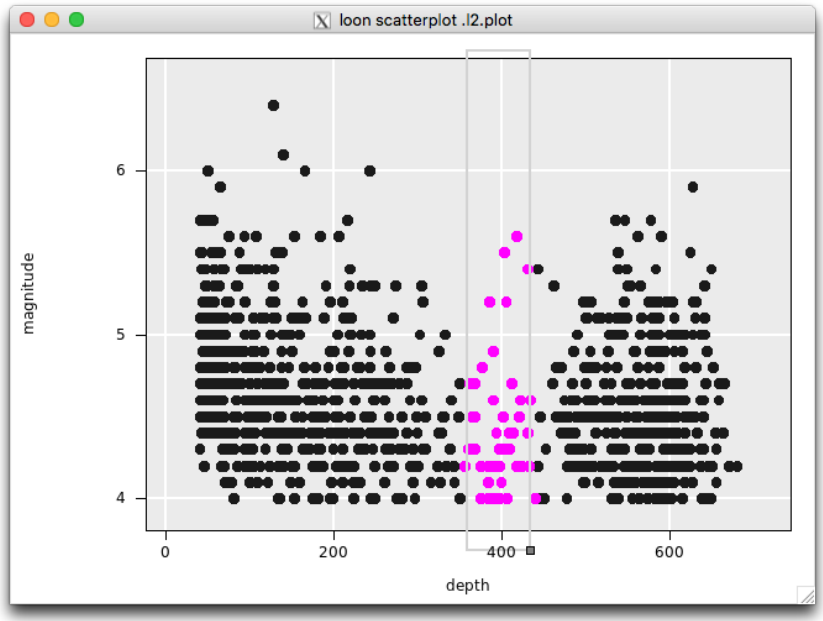


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

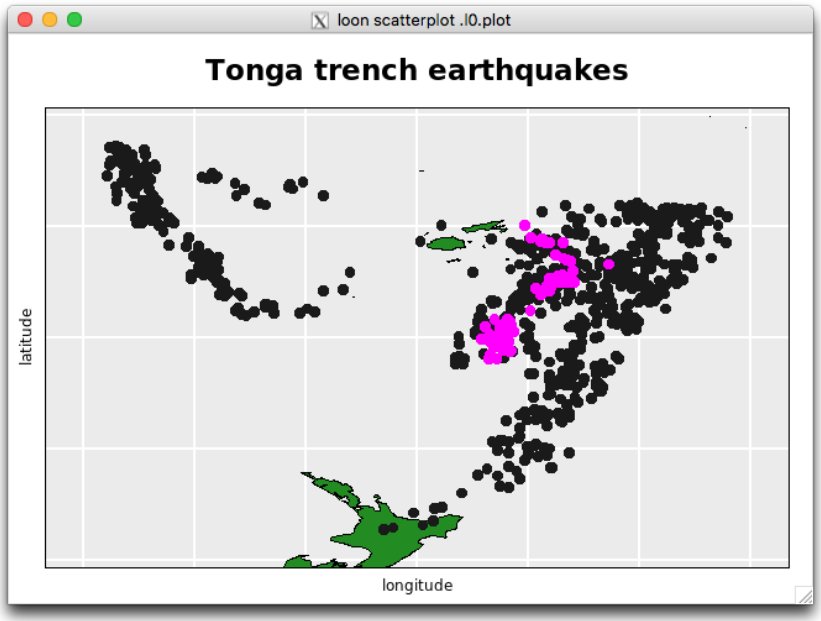


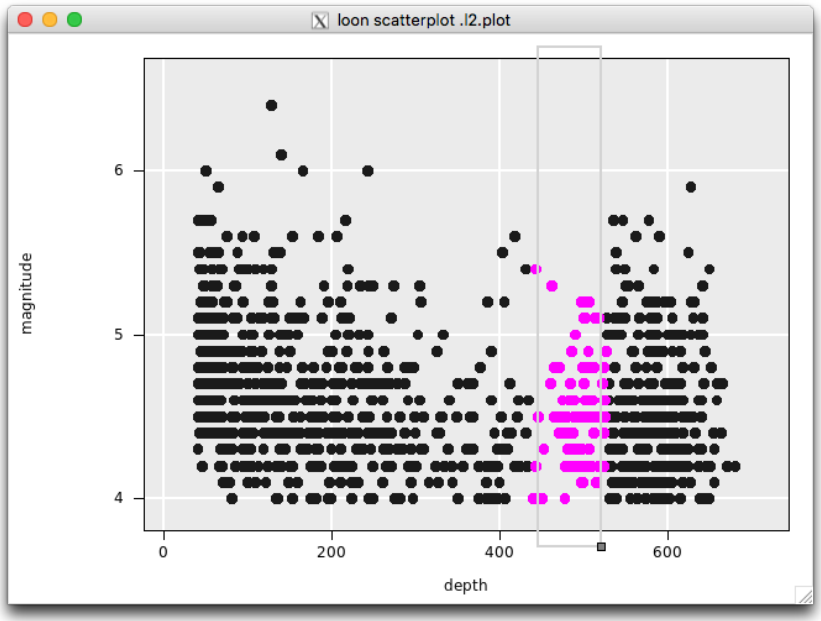


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

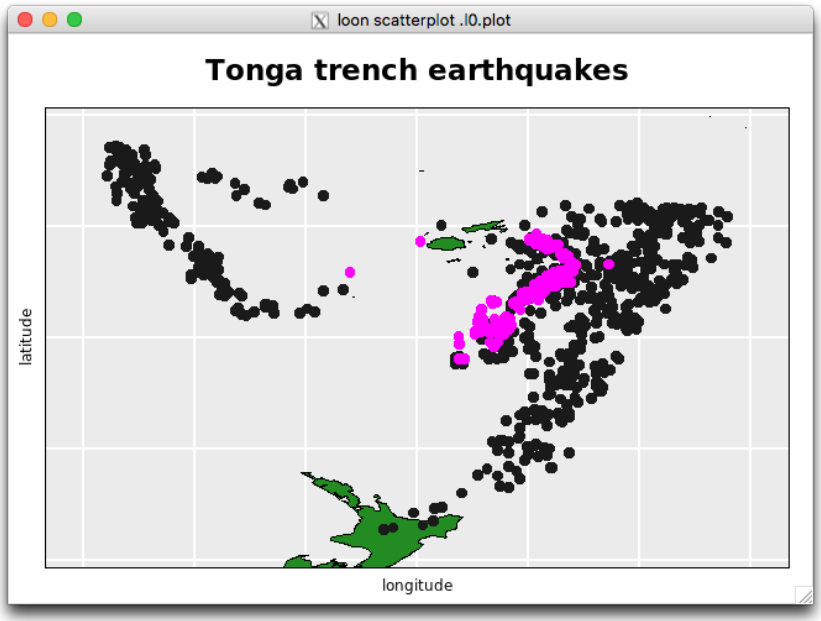


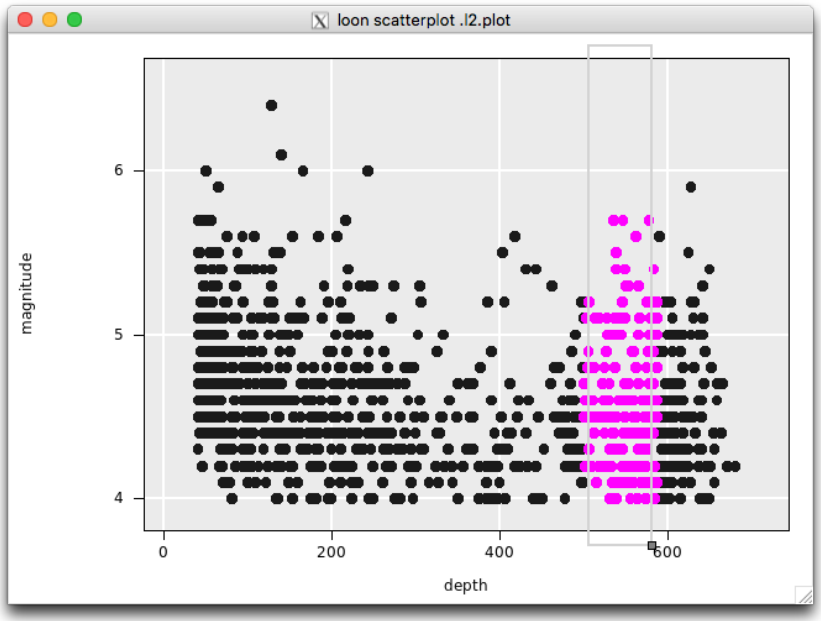


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

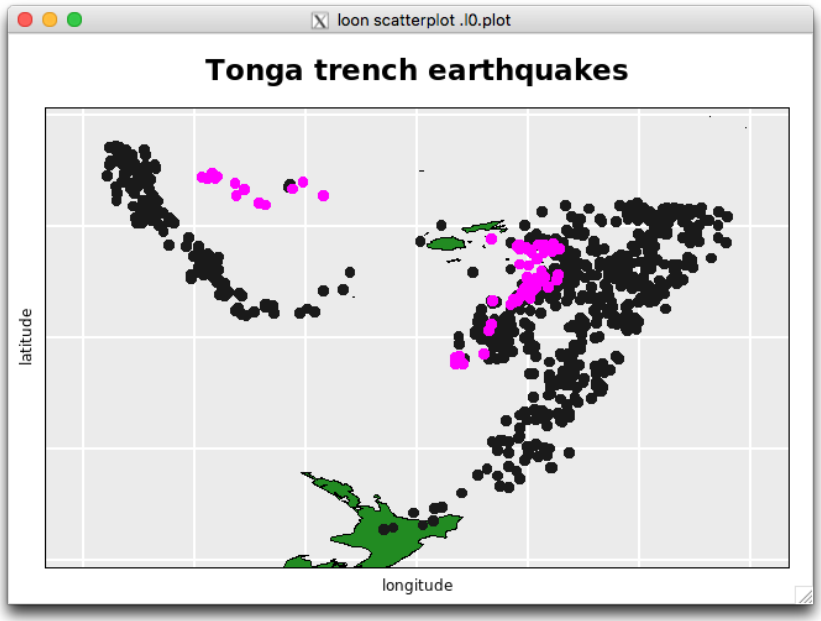


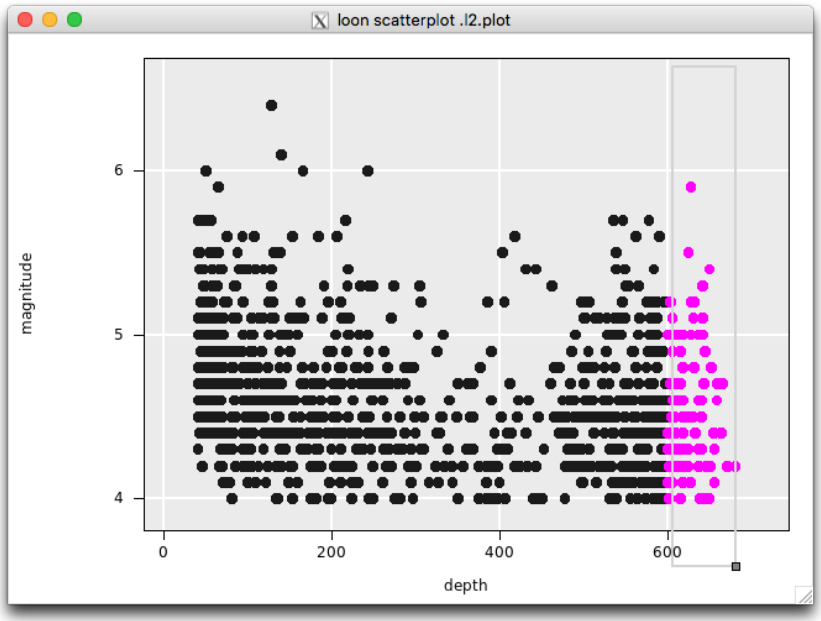


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:





We observe the spatial pattern at left as we brush various depths at right.

Loon plot states

A `loon` plot has only a single attribute, its class:

```
attributes(p)
```

```
## $class
## [1] "l_plot" "loon"
```

Nevertheless, there are various `tcl` states such as `color` that are associated with a `loon` plot. These may be queried and/or set programmatically.

Because these are `tcltk` data structures, some special functions have been written to access and set these states. These are

```
l_info_states(p)             # gives the names and values of the states on p
names(l_info_states(p))      # returns just the names (not their values)
l_cget(p, "linkingGroup")    # returns the value of the state "linkingGroup"
l_configure(p, color = "steelblue") # sets the color of all points to "steelblue"
```

Loon plot states

In R, loon provides some simpler means to achieve the same using more standard R functions. Namely, methods have been written to make the loon states more naturally accessible in R

```
names(p)            # returns just the names (not their values)
```

```
##  [1] "glyph"            "itemLabel"        "showItemLabels"   "linkingGroup"
##  [5] "linkingKey"       "zoomX"            "zoomY"            "panX"
##  [9] "panY"             "deltaX"           "deltaY"           "xlabel"
## [13] "ylabel"           "title"            "showLabels"       "showScales"
## [17] "swapAxes"         "showGuides"       "background"       "foreground"
## [21] "guidesBackground" "guidelines"       "minimumMargins"   "labelMargins"
## [25] "scalesMargins"    "x"                "y"                "xTemp"
## [29] "yTemp"            "color"            "selected"         "active"
## [33] "size"             "tag"              "useLoonInspector" "selectBy"
## [37] "selectionLogic"
```

```
p["linkingGroup"]   # returns the value of the state "linkingGroup"
```

```
## [1] "quakes"
```

```
p["color"] <- "steelblue"  # sets the value of the state linkingGroup to "quakes"
```

Note that for some states like "linkingGroup" more than one value needs to be set simultaneously and this is only achievable using `l_configure()`. For example,

```
# The following may fail when there are other plots in the new group
# because syncing info is also needed
p["linkingGroup"] <- "some other group"
# and this must be passed at the same time as the new group name; more on this shortly
l_configure(p, linkingGroup = "some other group", sync = "pull")
```

Linking - via `color`

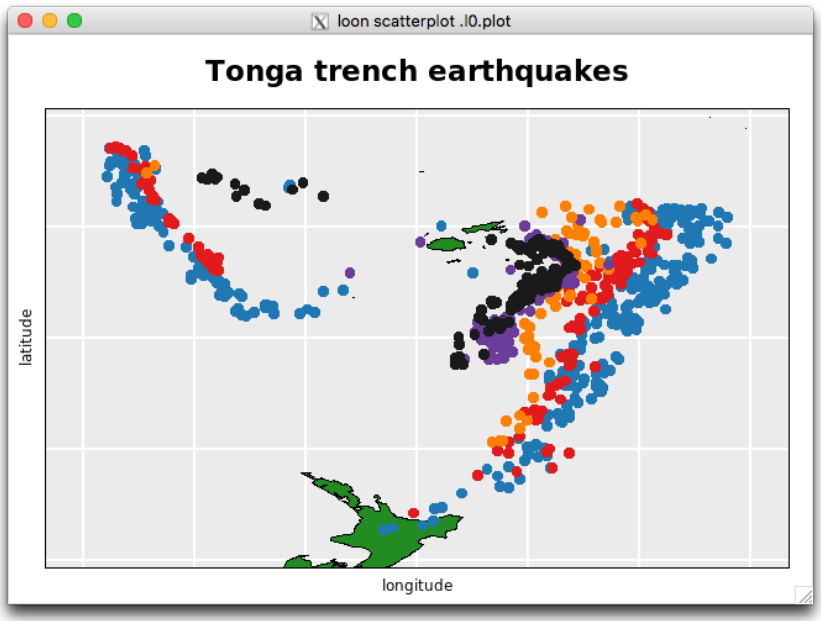
Rather than brush, we might also choose to colour the locations different colours according to the depth. This is easily accomplished either by selecting points and modifying their colour via the inspector, or by directly changing the `color` state of the plot `p`:

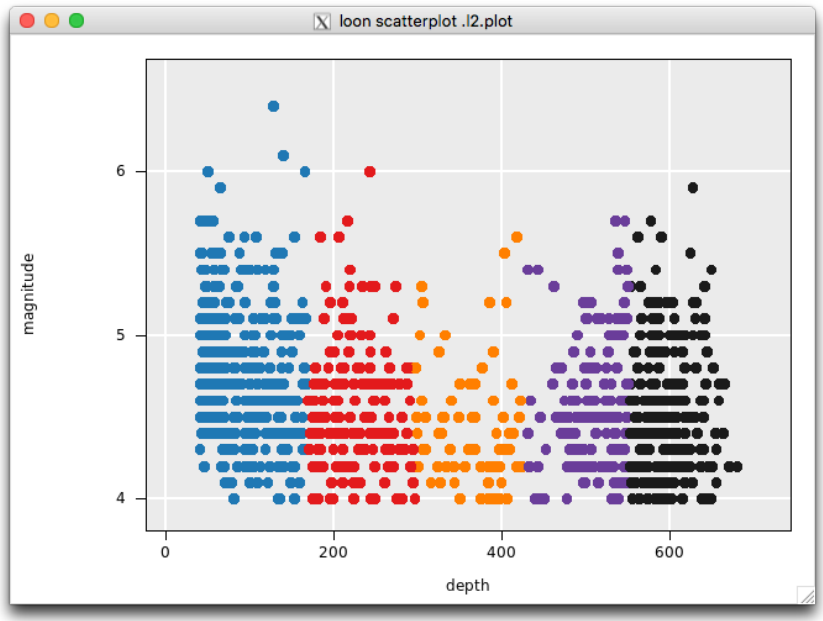
```
# get 5 (equal width) levels by cutting the depths up
depthLevels <- cut(quakes$depth, breaks = 5)
p['color'] <- depthLevels
```

Linking - via `color`

Rather than brush, we might also choose to colour the locations different colours according to the depth. This is easily accomplished either by selecting points and modifying their colour via the inspector, or by directly changing the `color` state of the plot `p`:

```
# get 5 (equal width) levels by cutting the depths up
depthLevels <- cut(quakes$depth, breaks = 5)
p['color'] <- depthLevels
```





Linking - via `color`

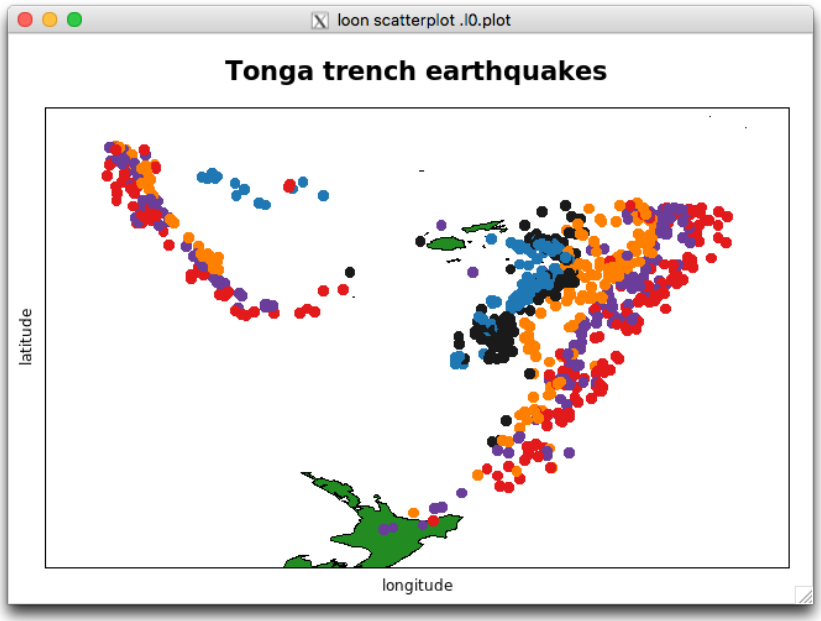
Note that equal count intervals can also be constructed by defining the break points:

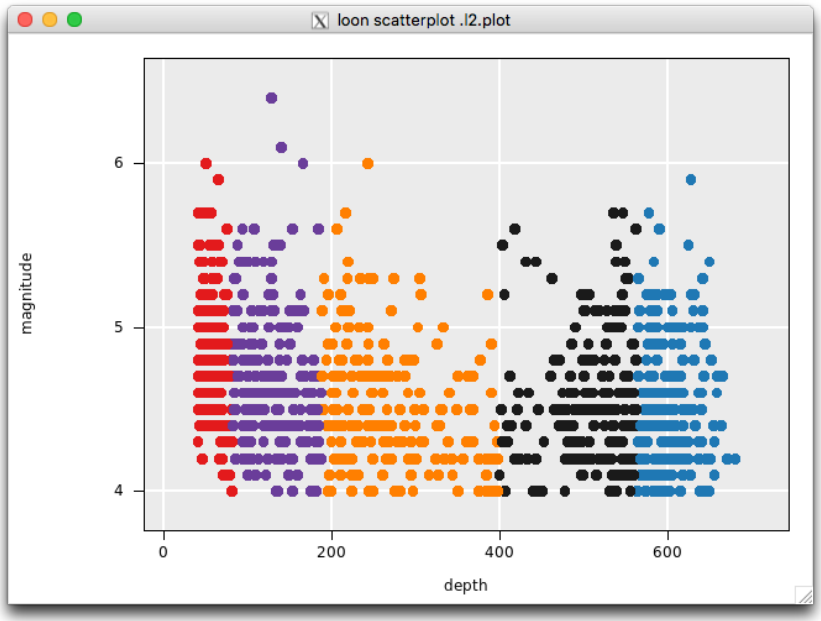
```
quantile_breaks <- quantile(quakes$depth, probs = seq(0, 1, 0.2))
quantile_breaks[1] <- quantile_breaks[1] - 1 # to counter minimum becoming NA
p['color'] <- cut(quakes$depth, breaks = quantile_breaks)
```

Linking - via `color`

Note that equal count intervals can also be constructed by defining the break points:

```
quantile_breaks <- quantile(quakes$depth, probs = seq(0, 1, 0.2))
quantile_breaks[1] <- quantile_breaks[1] - 1 # to counter minimum becoming NA
p['color'] <- cut(quakes$depth, breaks = quantile_breaks)
```





Linking - via `glyph` shape

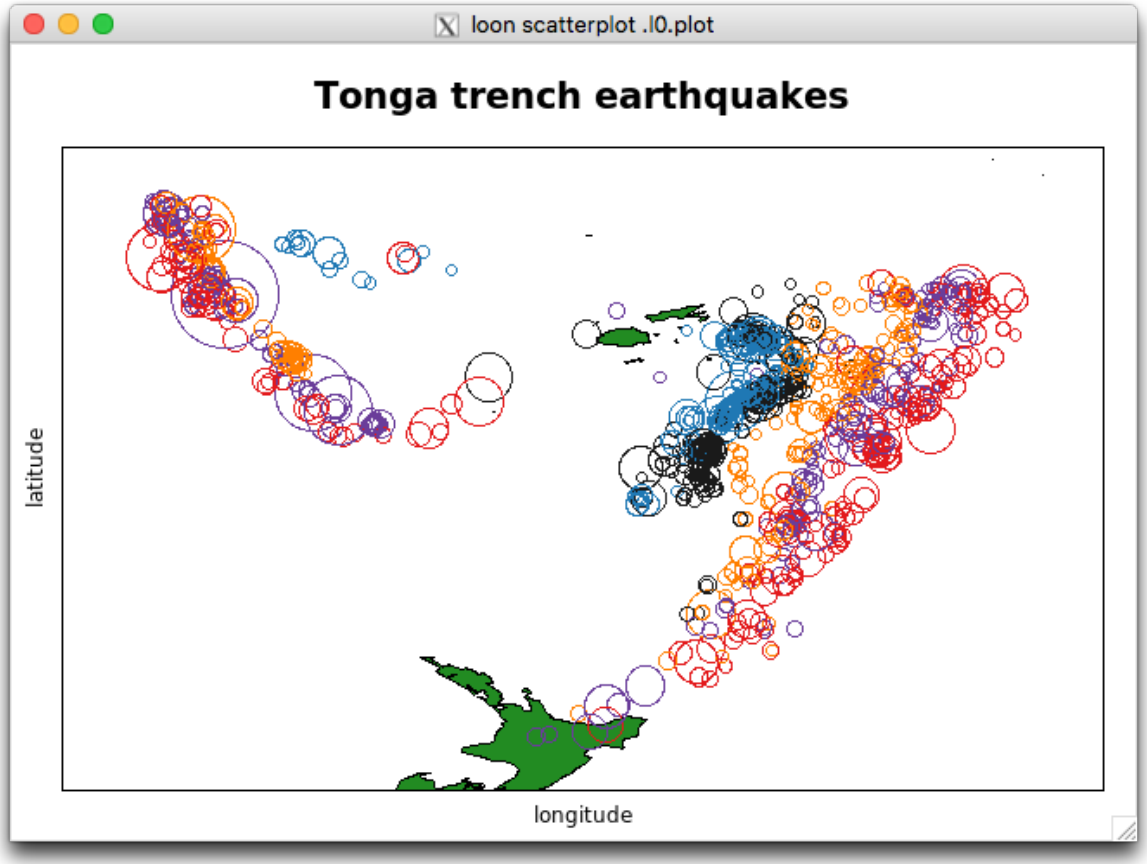
Similarly, we could change the shape of the points, that is its `glyph`, or its size programmatically:

```
p["glyph"] <- "ocircle"    # for "open circle"
sizeByMagnitude <-  (10 ^ quakes$mag) / 10000  # N.B. Richter scale is log scale
sizeByMagnitude <- 2 + sizeByMagnitude - min(sizeByMagnitude)
p["size"] <- sizeByMagnitude
```

Linking - via `glyph` shape

Similarly, we could change the shape of the points, that is its `glyph`, or its size programmatically:

```
p["glyph"] <- "ocircle"    # for "open circle"
sizeByMagnitude <-  (10 ^ quakes$mag) / 10000  # N.B. Richter scale is log scale
sizeByMagnitude <- 2 + sizeByMagnitude - min(sizeByMagnitude)
p["size"] <- sizeByMagnitude
```

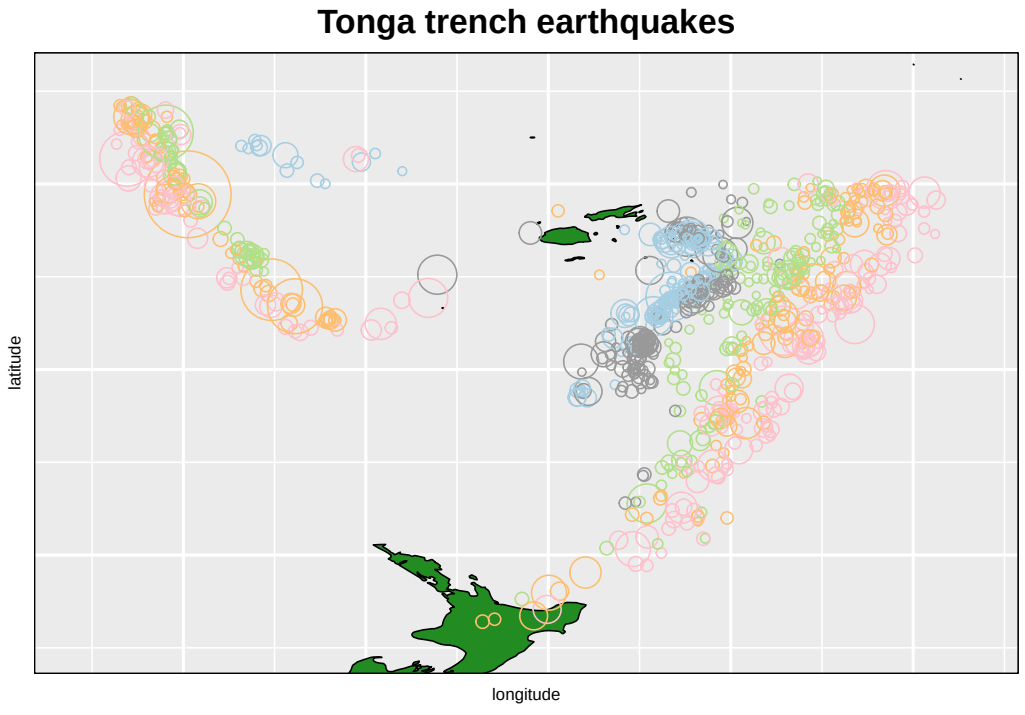


Printing the plot

Note again, that **at any time** the **current view** of the loon plot `p` can be transferred to a static `grid` graphics plot simply as `plot(p)`.

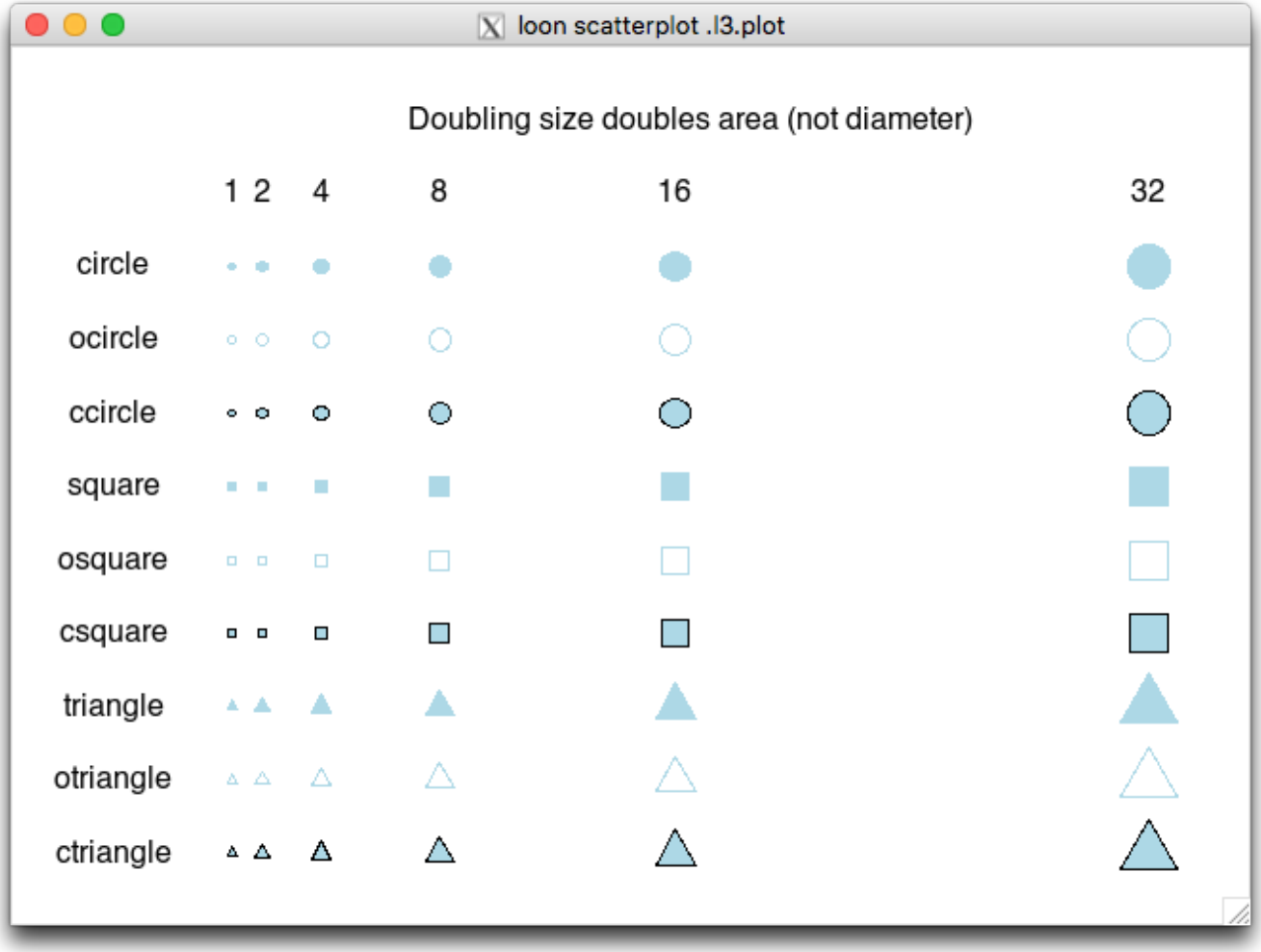
For example, `p` now looks like

```
plot(p)
```



Point glyphs - shapes and sizes

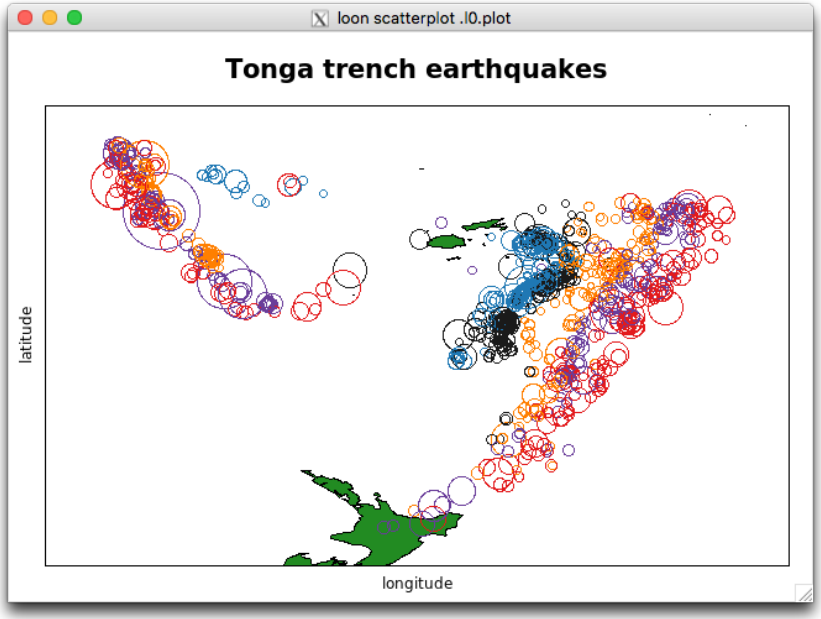
There are numerous different shapes (and sizes) to choose from:

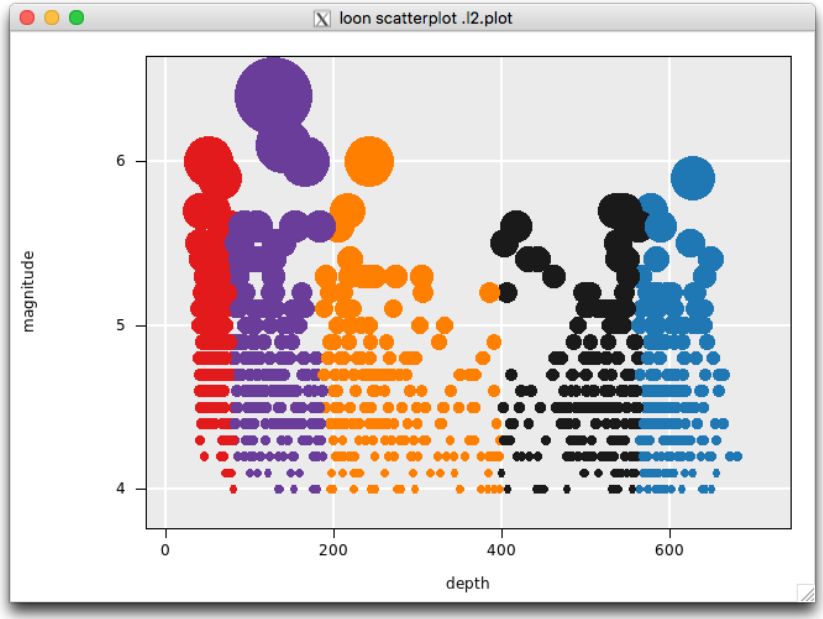


Again, `size` refers to the relative (for each shape) area of the glyph and does not mean its linear extent (or diameter). (To change the size dramatically, as if by linear extent, then the user might change the size by squaring.)

Linking - linked states

A glance at the second scatteplot shows that, like colour, the size of the glyph changed there because it was linked to the first scatterplot. However the shape of the glyph did **not**.





The states which are linked to, and hence eligible to change with, the plots participating in the same `linkingGroup` are unique to each plot:

```
l_getLinkedStates(p)
```

```
## [1] "color"    "selected" "active"   "size"
```

Linking - linked states

Plots in the same `linkingGroup` indicate which of their states, they are willing to have change with the group. These linked states can be queried via `l_getLinkedStates()` as before:

```
l_getLinkedStates(p)
```

```
## [1] "color"    "selected" "active"   "size"
```

Linked states can be updated for any plot using `l_setLinkedStates()` as in:

```
l_setLinkedStates(p, c(l_getLinkedStates(p), "glyph"))
l_setLinkedStates(p2, c(l_getLinkedStates(p2), "glyph"))
```

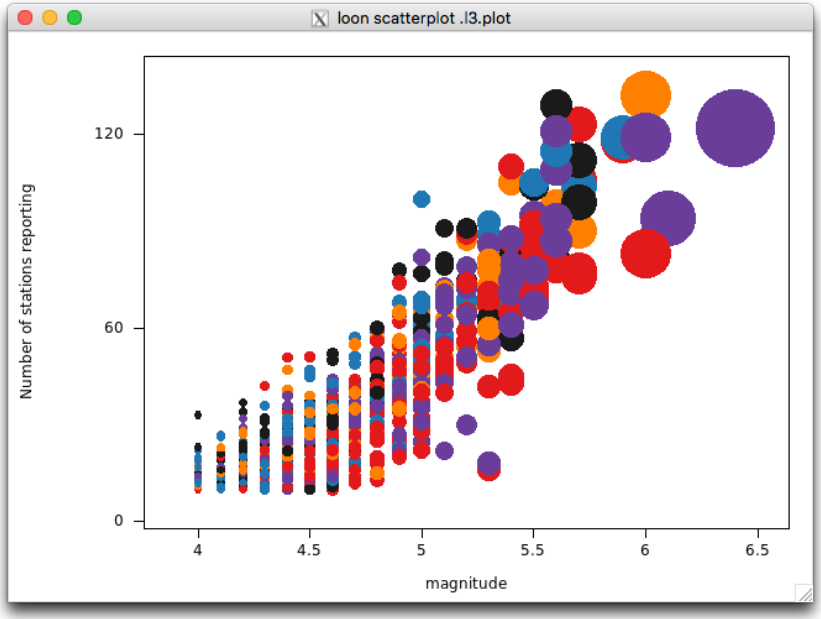
As a consequence each of `p` and `p2` will now change its `glyph` whenever the other does (note that this requires the glyph to actually change before that change is propagated). This is because they are members of a common `linkingGroup` quakes, which can be queried (and set) via `p['linkingGroup']`.

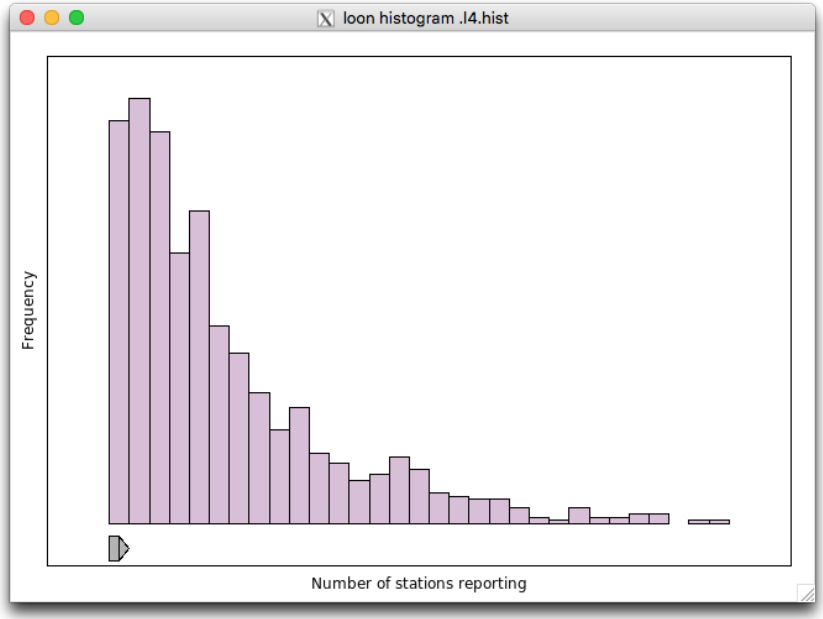
Should another plot, say `p3`, be added to the same `linkingGroup`, then the state `glyph` for `p3` will not change whenever that of either `p` or `p2` does unless `p3` has also registered its `glyph` state as a linked one. Only those linked states that are common between plots in the same `linkingGroup` will be linked.

Plots and histograms via `l_hist()`

We could introduce other plots that are linked to the previous plots

```
p3 <- l_plot(quakes$mag, quakes$stations,
             xlabel = "magnitude", ylabel = "Number of stations reporting",
             showScales = TRUE, linkingGroup = "quakes")
h <- l_hist(quakes$stations, xlabel = "Number of stations reporting",
            linkingGroup = "quakes")
```



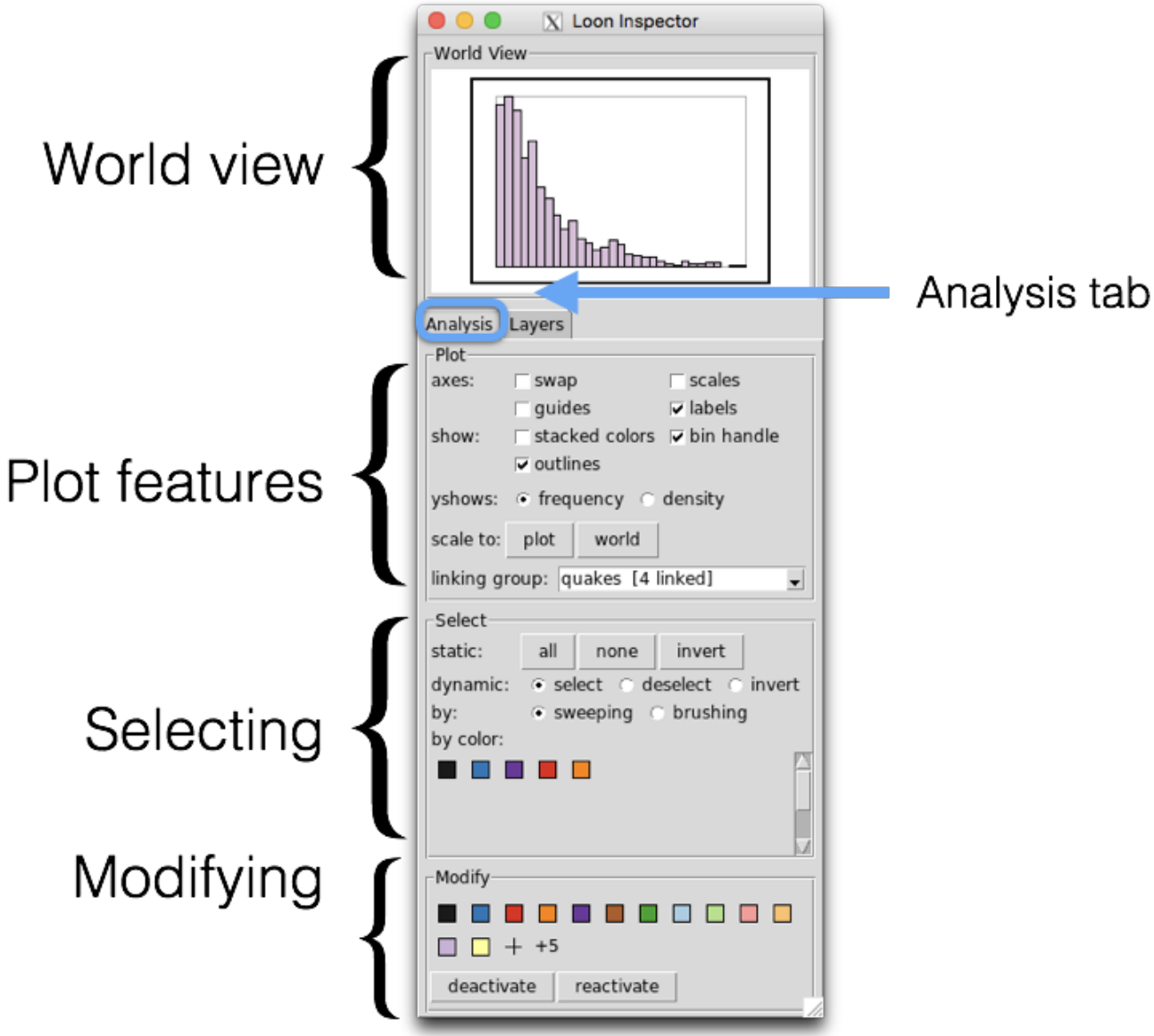


Note:

- the strong positive relationship between the number of stations reporting a quake and its magnitude
- the size of the circle is taken from the common `linkingGroup` but the shape is not
- the colours associated with depth appear in the scatterplot but not in the histogram

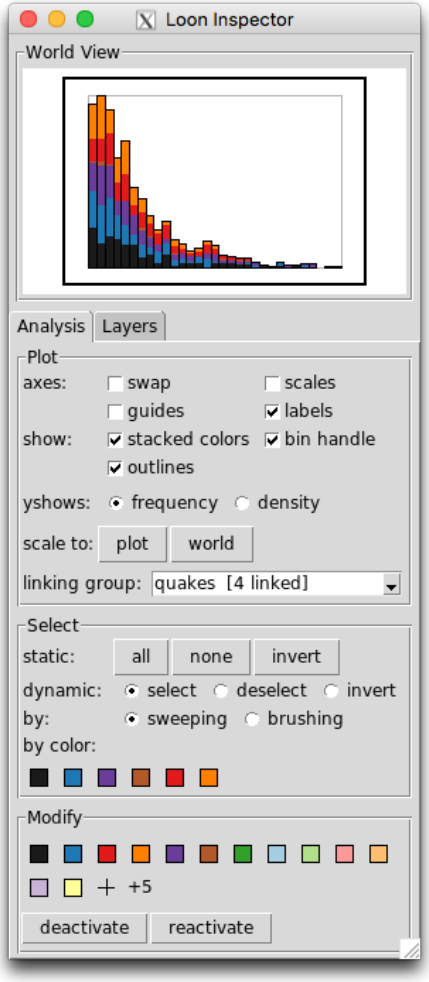
Loon inspector for a histogram - Analysis tab

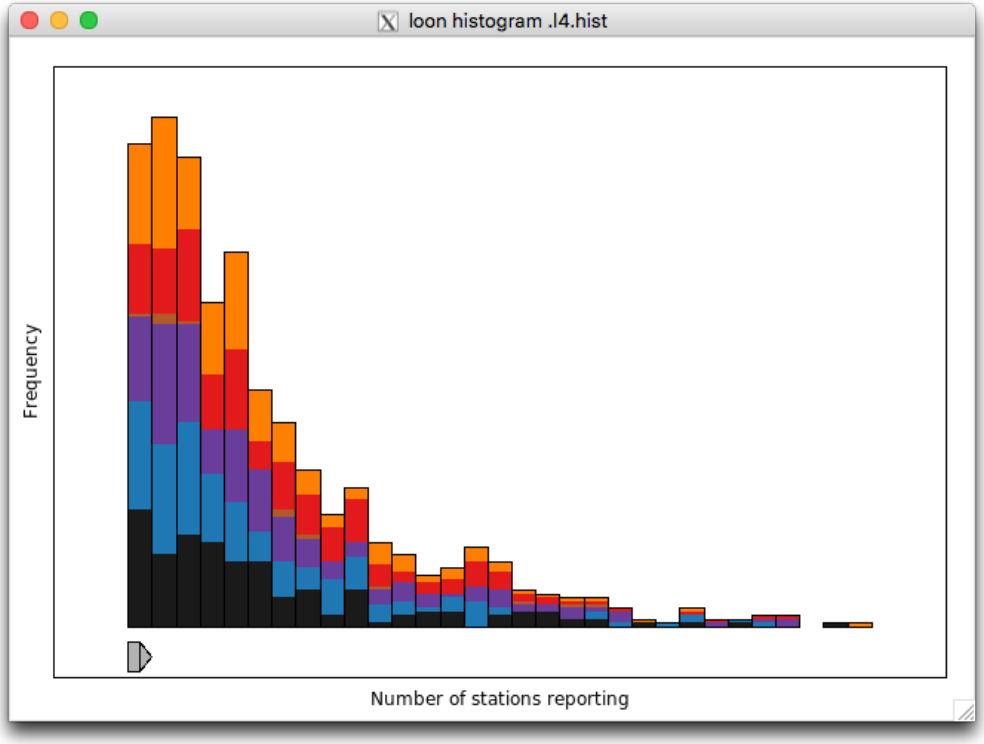
When the histogram is the active window, the inspector changes its focus to the histogram, now with panels specialized for a histogram on the analysis tab:



Analysis tab - `stacked colors` check box

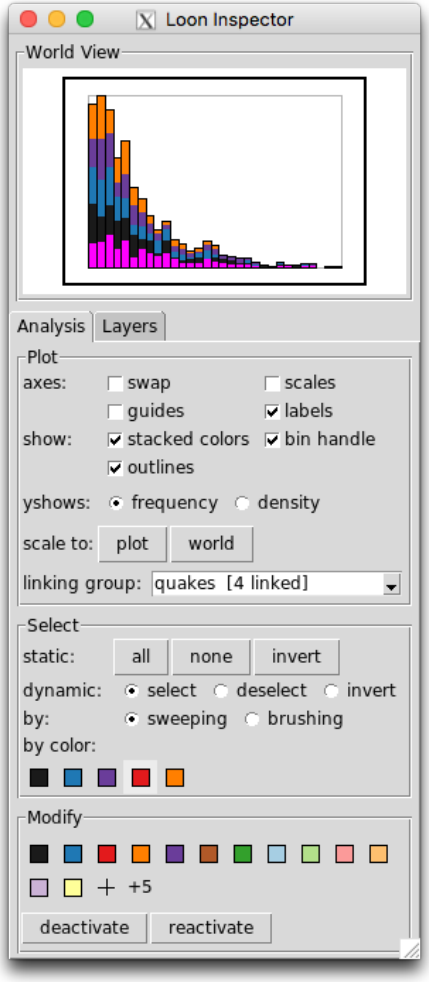
To see where the various groups fall in the histogram, check "stacked colors" box in the plot options section on the "Analysis" tab:

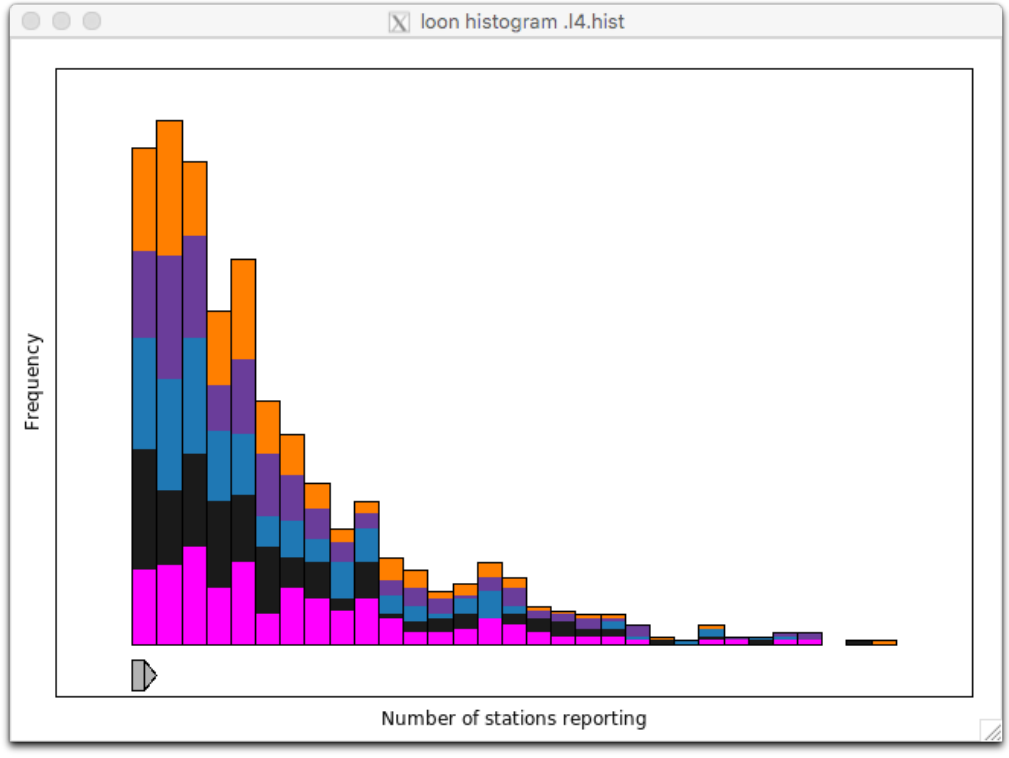




Analysis tab - selection by colour

Selecting by colour will drop the selected colours to the bottom of the histogram.





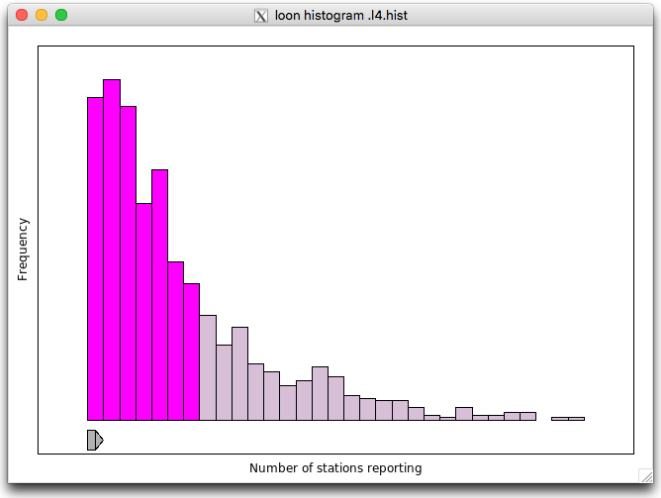
Multiple selection of colours is possible (i.e. via "shift + select") from the inspector, or directly on the histogram itself.

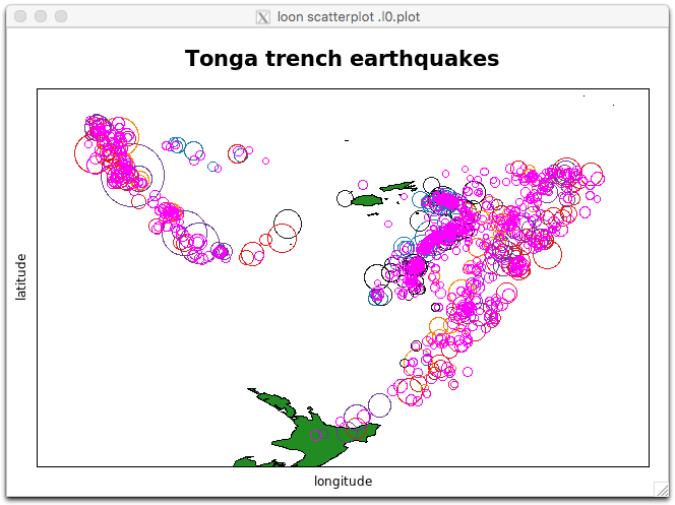
In the latter case, the colours are selected within each bar.

Histogram - selection by colour

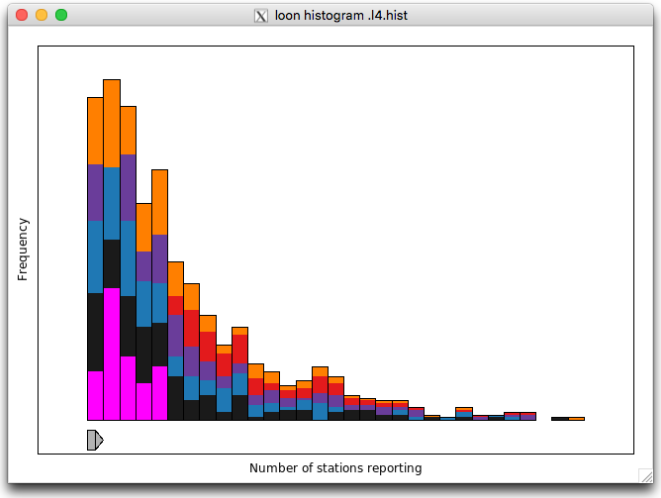
Bars (and colours when stacked) can be selected on the histogram itself:

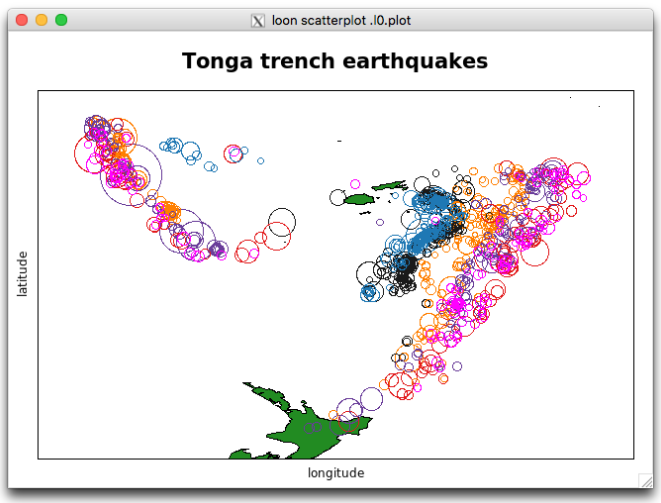
Selecting bars





Selecting colours (a few reds)





Three dimensional scatterplots –

The quakes data has three spatial variables `lat`, `long`, and `depth`, which would be natural to view in a three dimensional plot.

First, because these are three very different scales (degrees latitude and longitude, and kilometres deep) the data needs to be rescaled. The function `l_scale3D()` rescales the variables of its argument to a more nearly common scale.

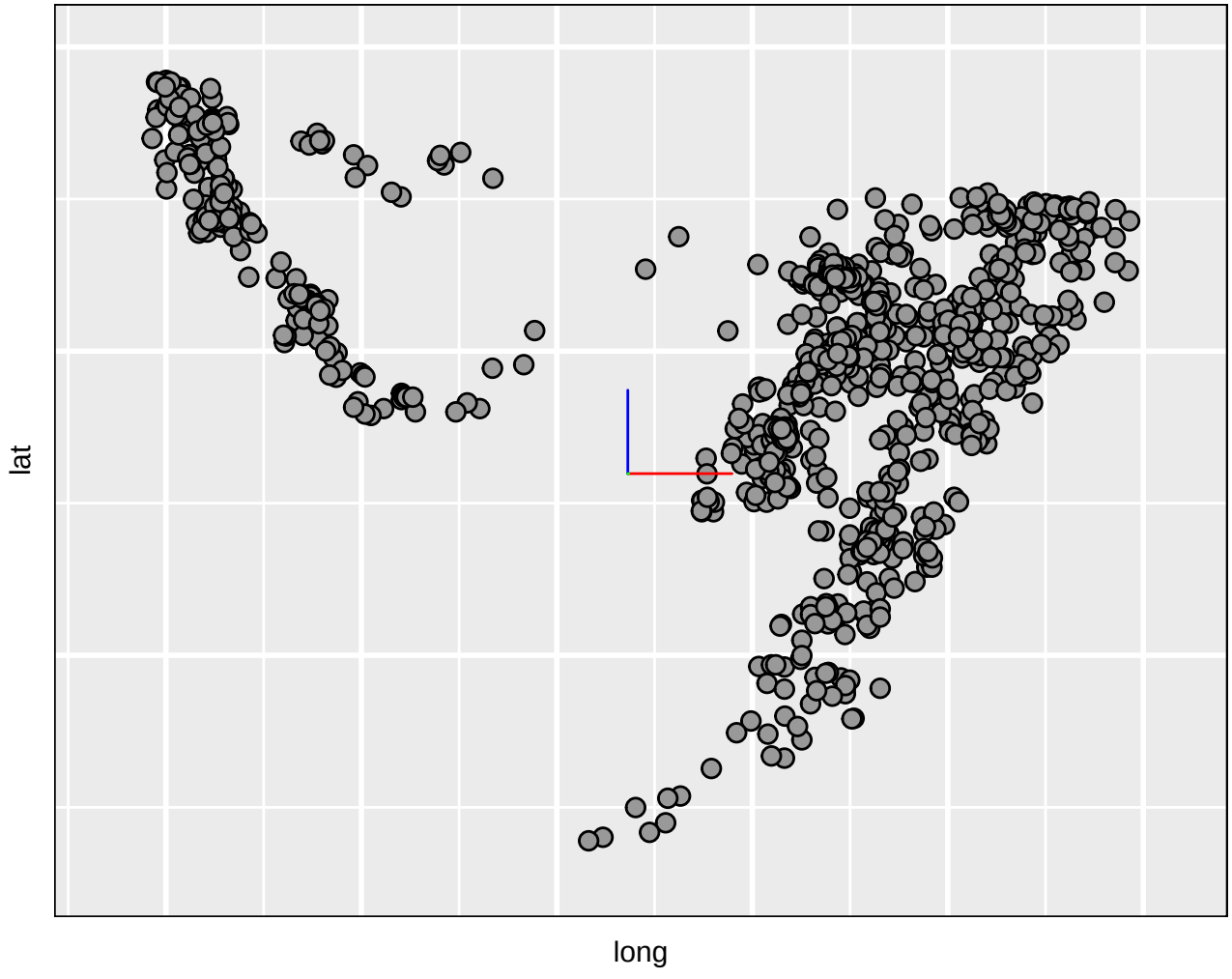
The rescaled data can then be passed to `l_plot3D()` to produce a three dimensional scatterplot.

```
p3D <- with(l_scale3D(quakes),
            l_plot3D(x = long, y = lat, z = depth)
            )
#  Coulr make axis bigger with
#  p3D["axisScaleFactor"] <- 4
```

Three dimensional scatterplots –

The 3D plot looks much like the 2D scatterplot but with an axis shown in colour.

```
plot(p3D)
```



The R key **toggles** rotation mode on and off. Arrow keys rotate as does the left mouse. Right mouse changes the origin of rotation.

Printing a plot - wysiwig or not?

Because `loon` plots are constructed in `tcltk` and their static versions in `grid`, there may be small discrepancies between the two versions (typically in size and font determinations) caused by translations from one system to the other.

These might could be addressed in at least three ways:

1. Adjust the `loon` graphic (e.g. changing glyph sizes) to effect the desired change in the consequent `grid` graphic.
2. Adjust the `grid` object (or `grob`) itself. The translation (and drawing) is carried out by the `loon` function `grid.loon()`, as in

```
gp <- grid.loon(p, draw = FALSE)
gp
```

```
## gTree[GRID.gTree.66]
```

   which produces a `grid` graphics `gtree` containing all the information needed to plot (and change) the translated `loon` plot in `grid`.
3. Most (all?) modern operating systems have a "screen shot" capability that allows (at least) a pixmap image of any window to be captured. This could be used on any `loon` window (including the inspector) and in fact was used to produce most of the `loon` images in this document.

Printing a plot - more on the `grid` connection

Graphics in the `grid` package are built up from graphical objects or `grobs`.

In `loon`, in addition to simply `plot(p)`, functions `grid.loon()` and `loonGrob()` can be used to construct `grobs` from `loon` plots that can in turn be used as any other `grob` in `grid`.

- `grid.loon(p)` translates, draws, and (invisibly) returns the `grob` corresponding the current state of the loon plot `p`.
- `loonGrob(p)` translates and returns the `grob`
- `plot(p)` translates and draws the `grob`

The resulting grob, can be used with all the rich functionality of the `grid` package. For example,

```
library(grid)
grid.ls(gp)  # lists the contents of the grob
# Or, in RStudio, can be viewed interactively
View(gp)
```

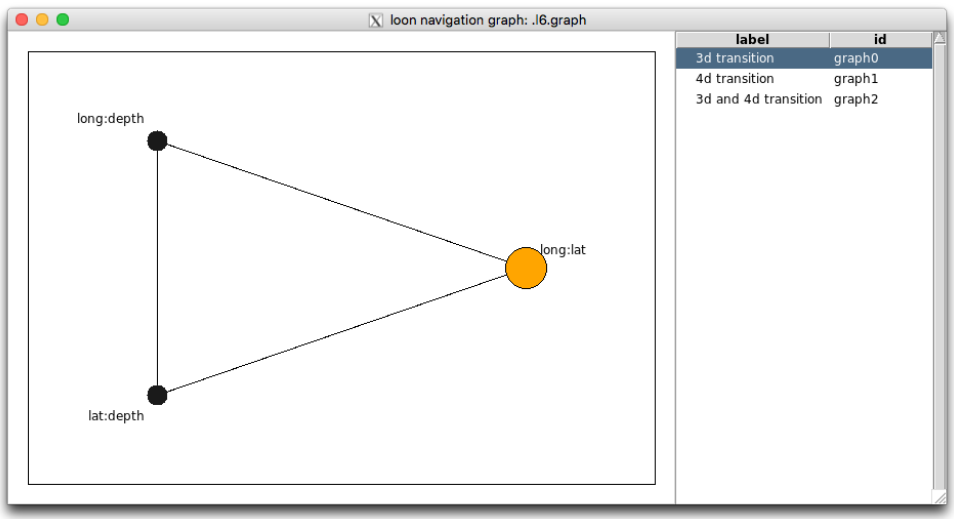
Note that all of the elements of a loon plot appear in the grob, either explicitly or, if they were not drawn in the loon plot, as an empty `grob` containing the arguments relevant to drawing them.

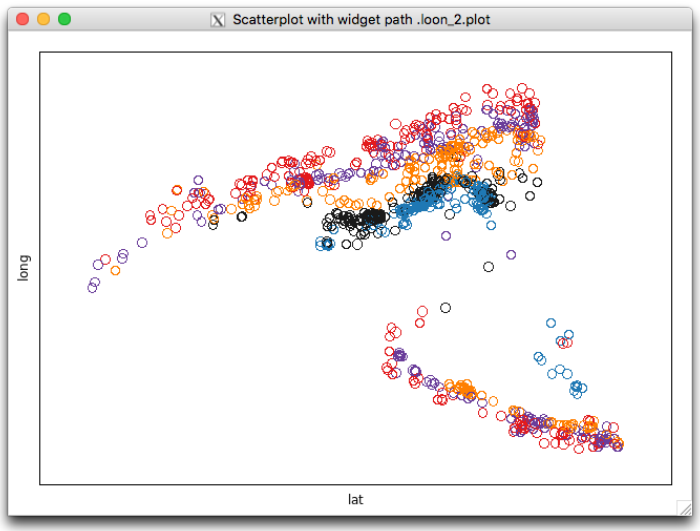
Being able to get and set the plot contents means that it is possible to capture different plot states as `grobs`.

Navigation graphs

`loon` also provides a simple means to examine three (and higher) dimensional structure interactively via its concept of a "navigation graph":

```
# Get a navigation graph (set all sizes to be the same)
ng <- l_navgraph(quakes[,c("long", "lat", "depth")],
                 linkingGroup = "quakes", sync = "pull",
                 glyph = "ocircle", size = 5)
# Note that specifying the size will propagate this throughout
# the linking group OVERRIDING the sync = "pull" argument.
```





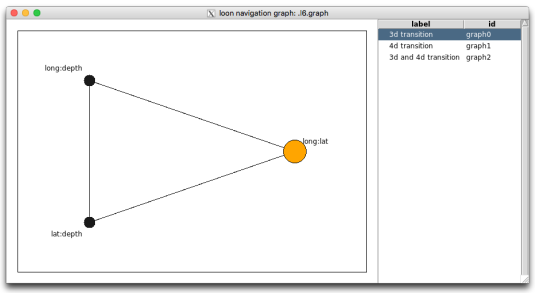
Nodes represent scatterplots of the named variates, edges are 3d transitions found by rotating around the axis of the shared variate.

The large coloured circle, called the "navigator", identifies the scatterplot shown in the separate display at right.

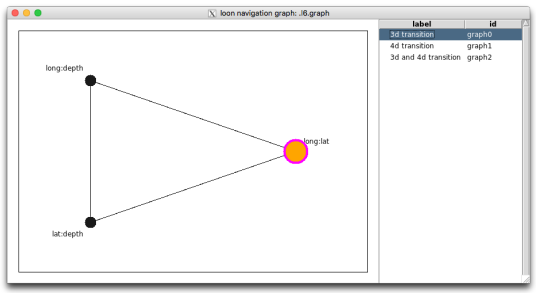
Navigation graphs - the graph navigator

Selecting the navigator highlights which nodes are connected to it. Then one is shift-selected to identify which scatterplot is to appear next.

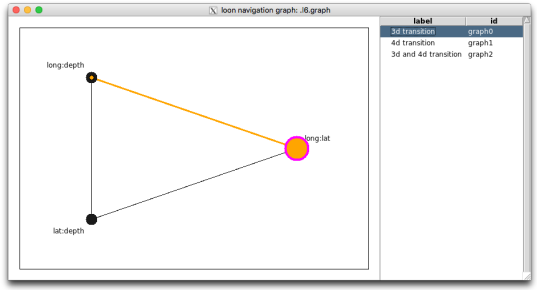
Navigator not selected.



Navigator selected.



Destination node selected.

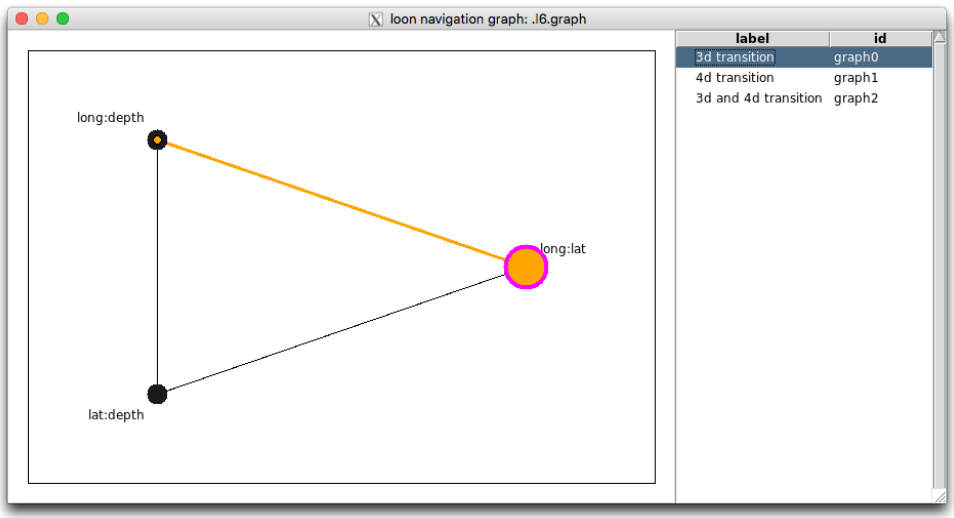


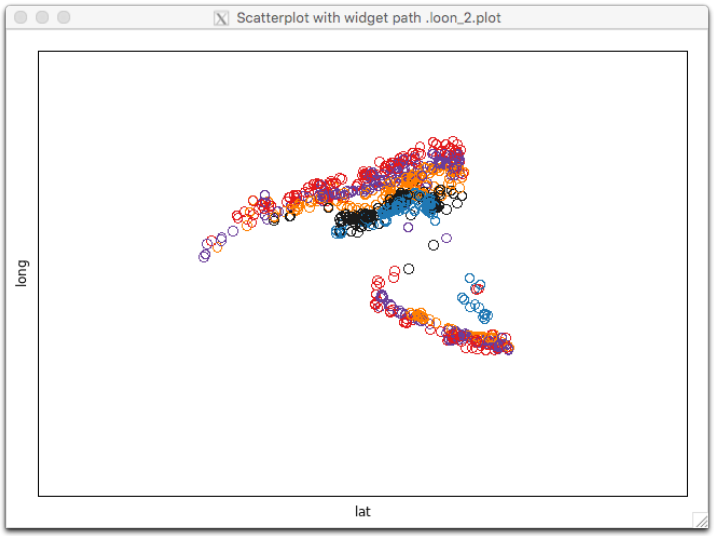
This can continue throughout the graph to produce a path through the three dimensional space.

- Connected nodes will highlight at each step.
- The navigator needs to be moved off any node that is to be selected next.
- The navigator can be dragged or moved using scrolling.
- Each edge represents a 3d transition from one 2d space (scatterplot) to the next.

Navigation graphs - navigating along a path

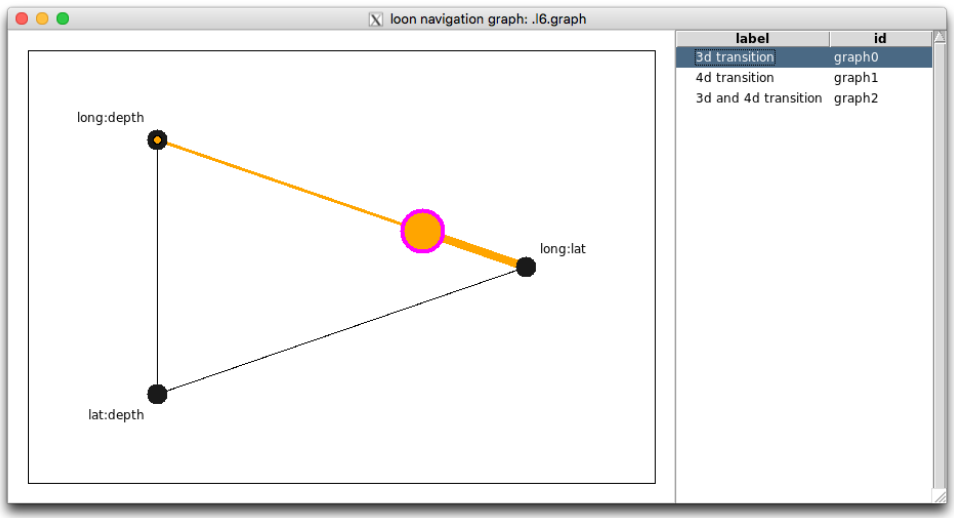
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

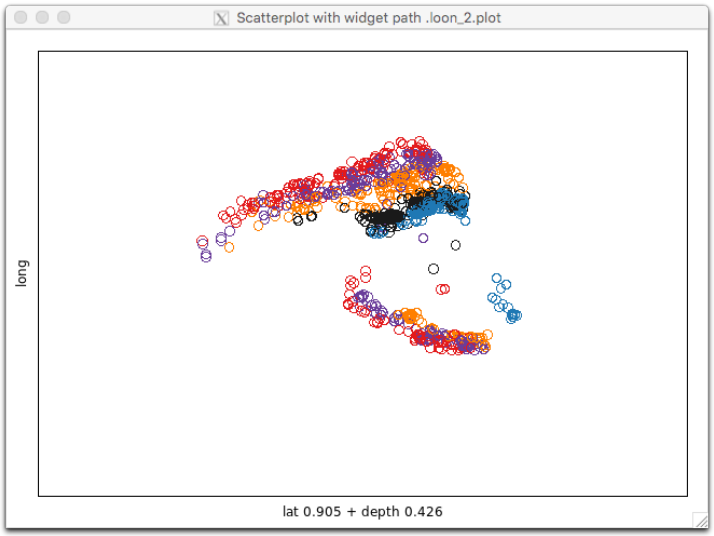




Navigation graphs - navigating along a path

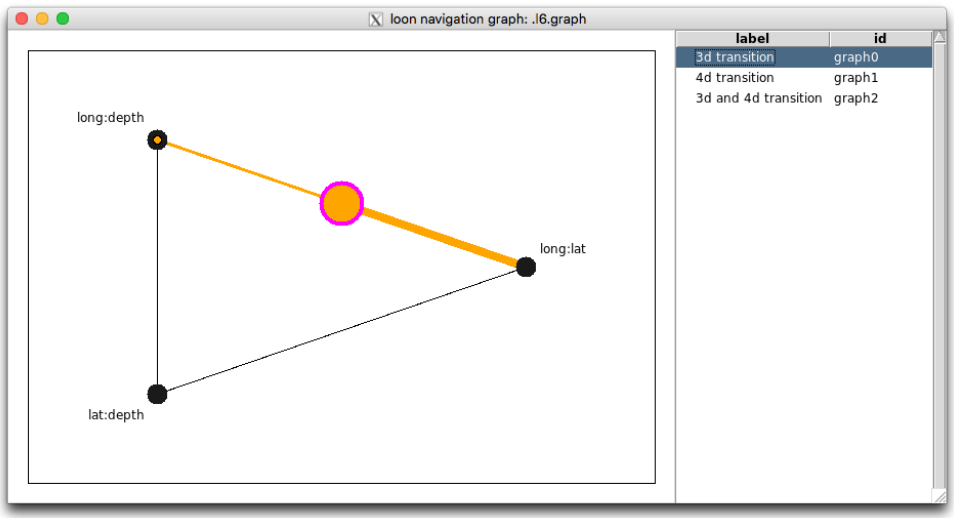
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

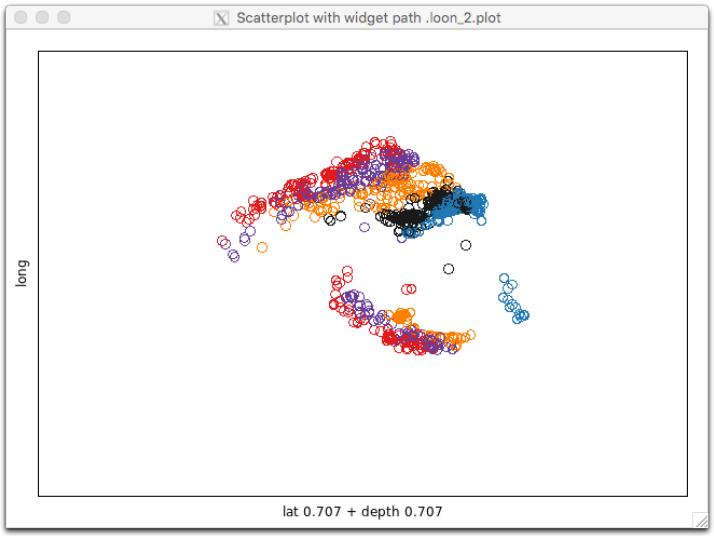




Navigation graphs - navigating along a path

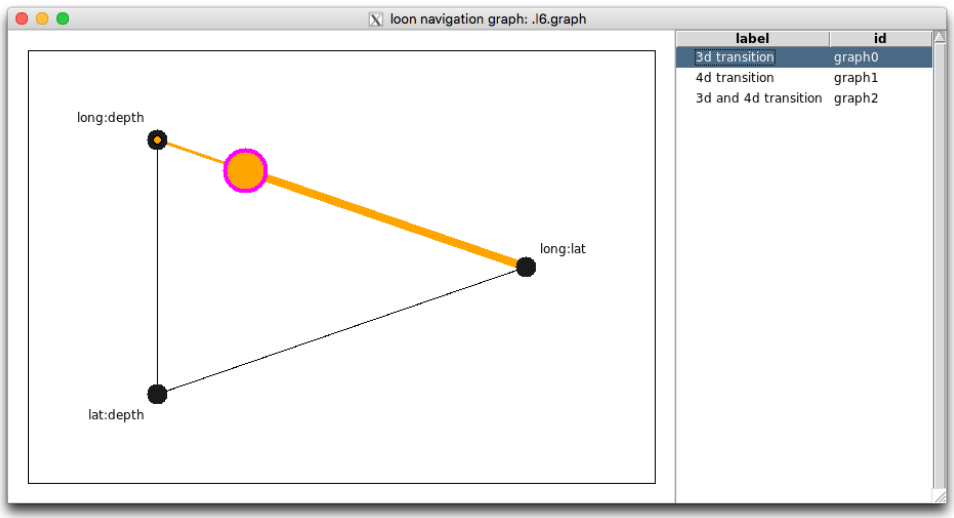
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

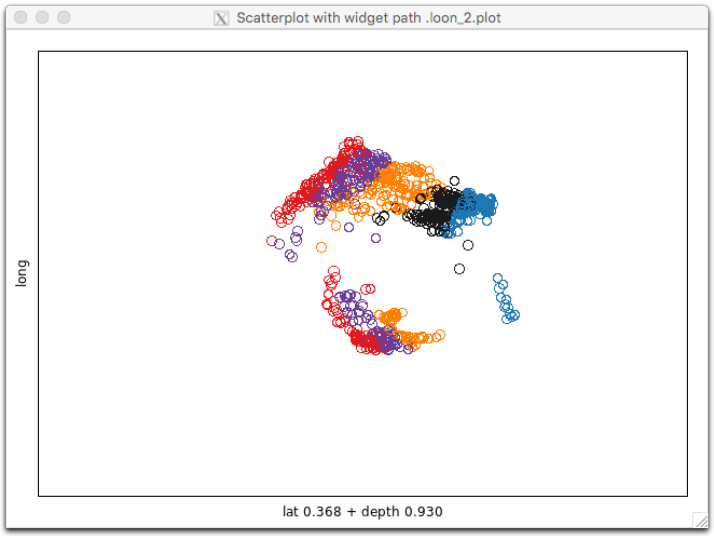




Navigation graphs - navigating along a path

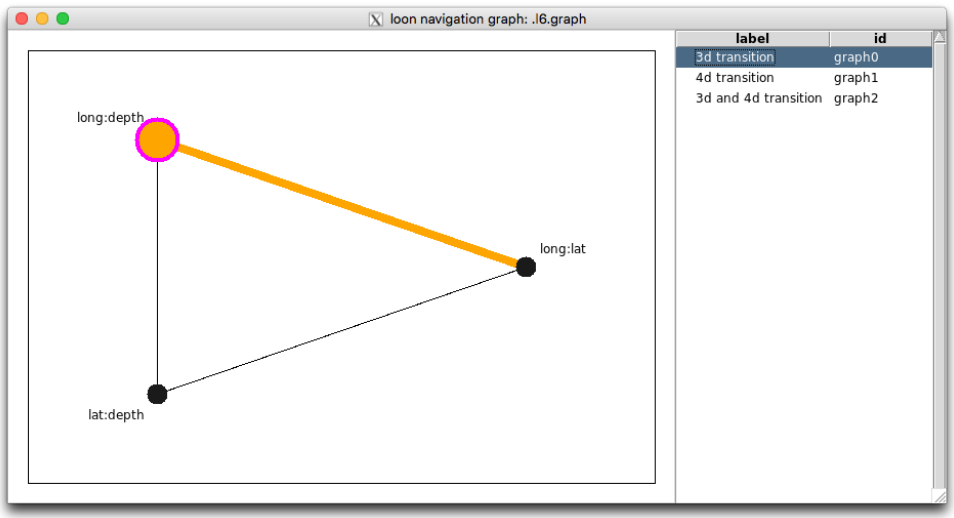
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

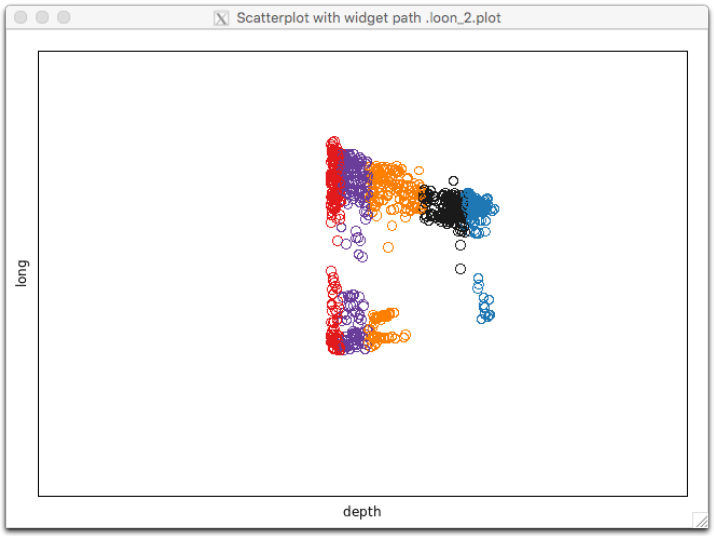




Navigation graphs - navigating along a path

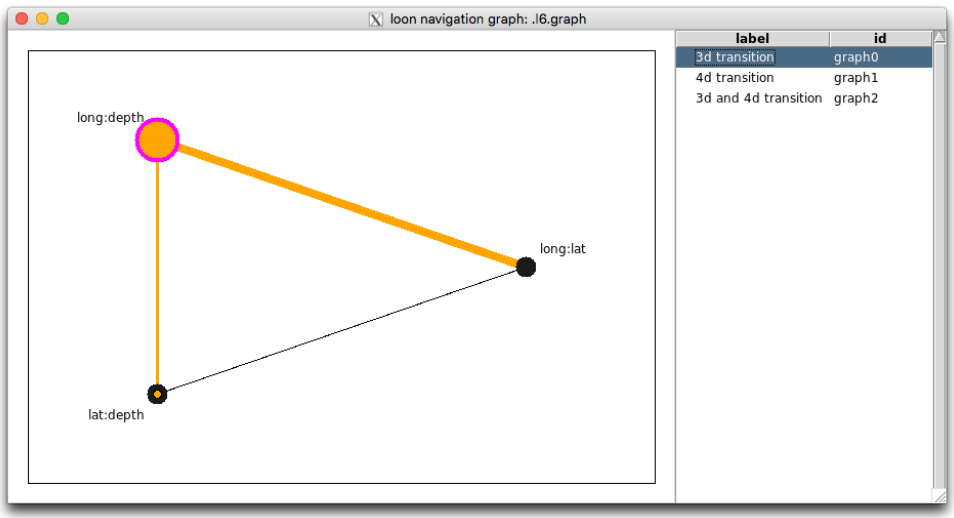
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

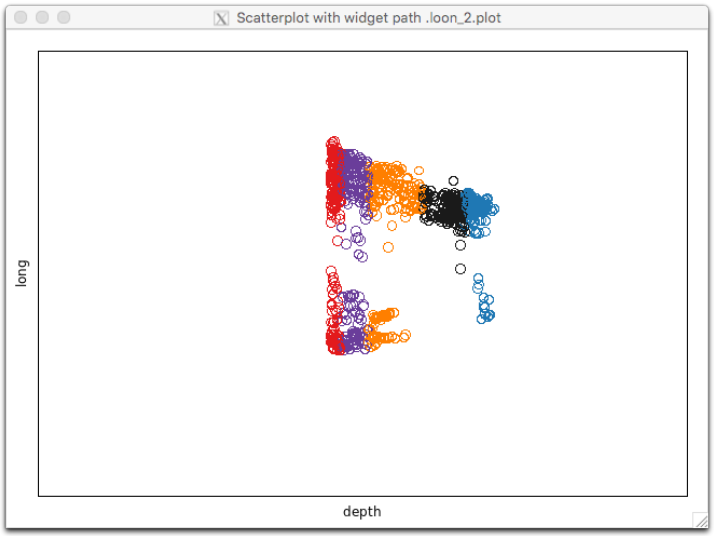




Navigation graphs - navigating along a path

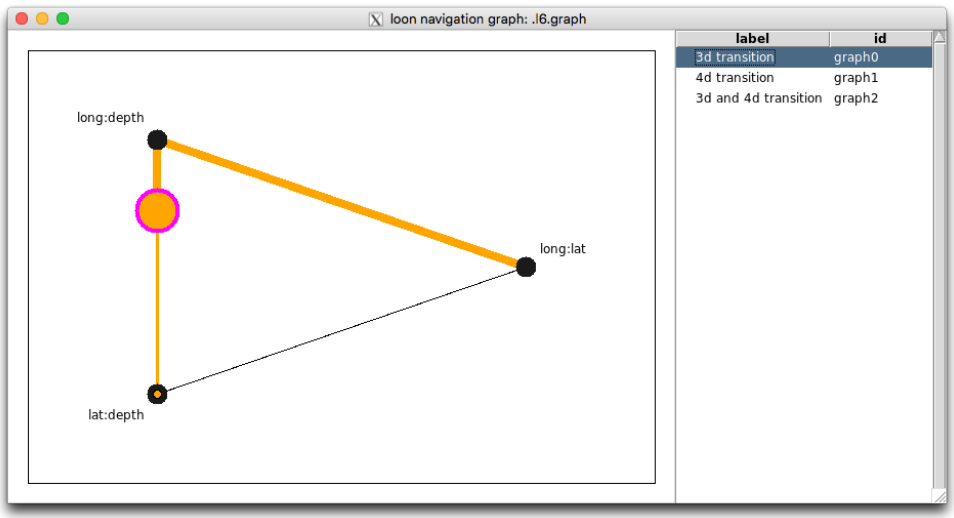
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

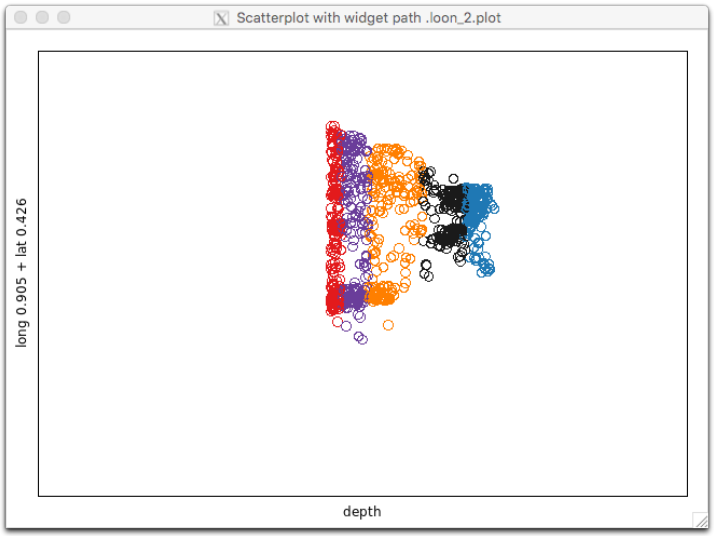




Navigation graphs - navigating along a path

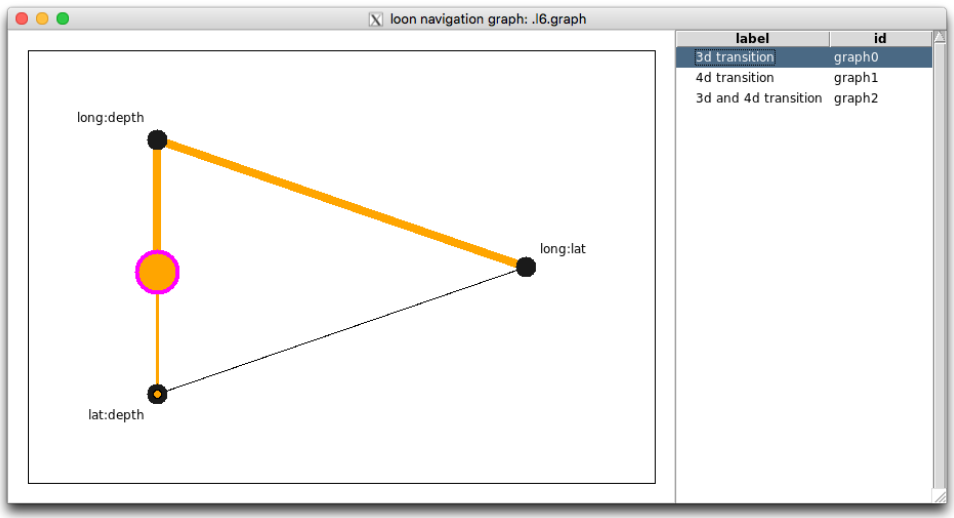
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

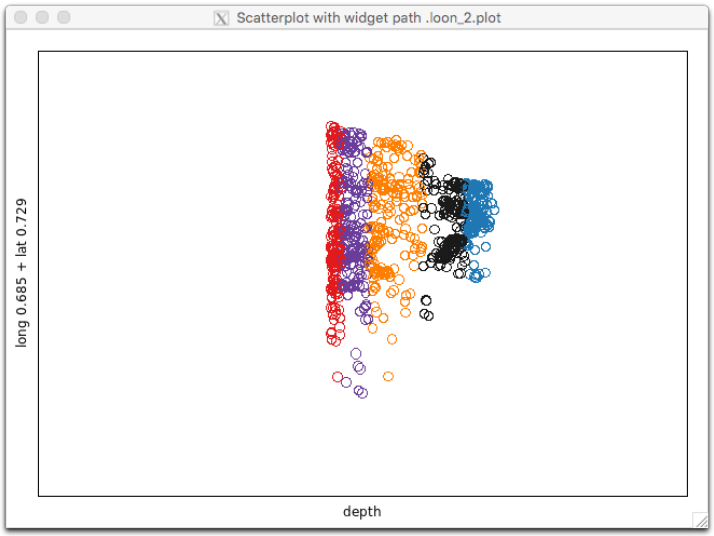




Navigation graphs - navigating along a path

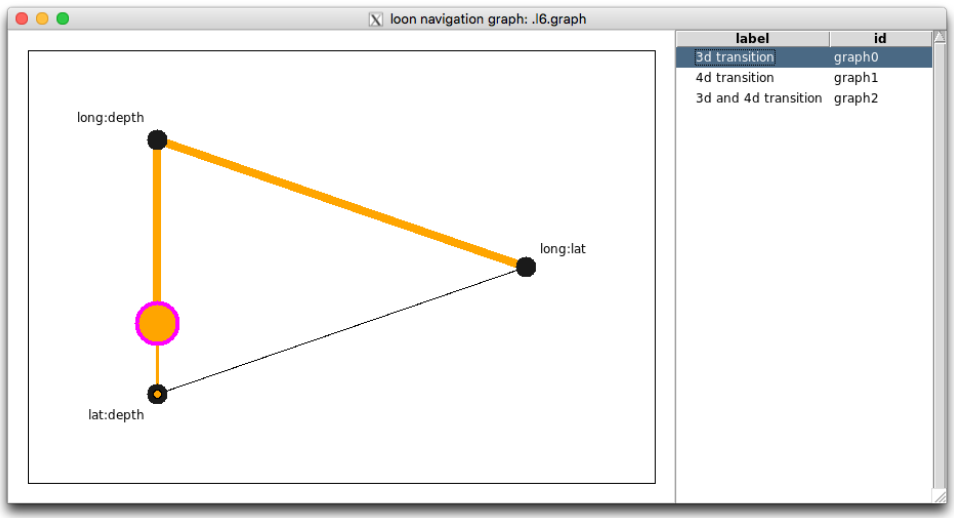
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

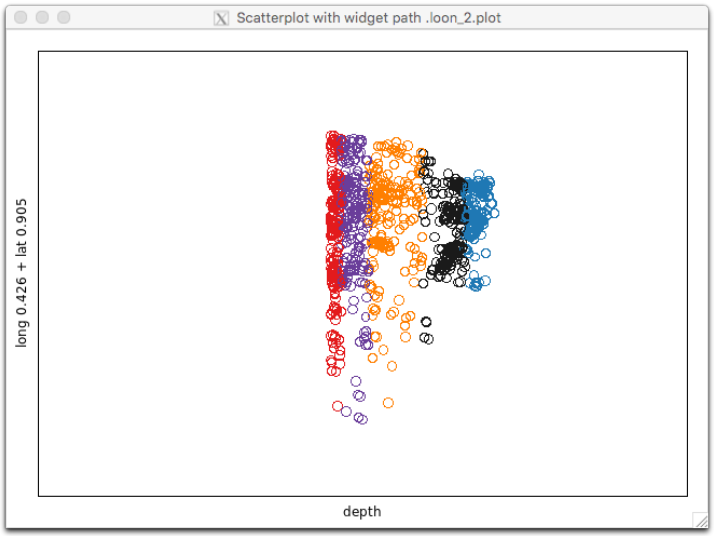




Navigation graphs - navigating along a path

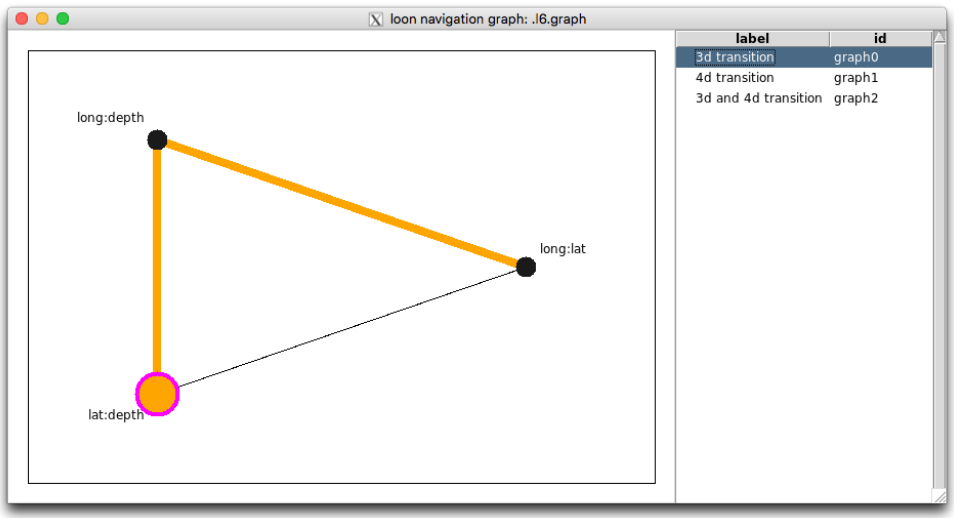
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

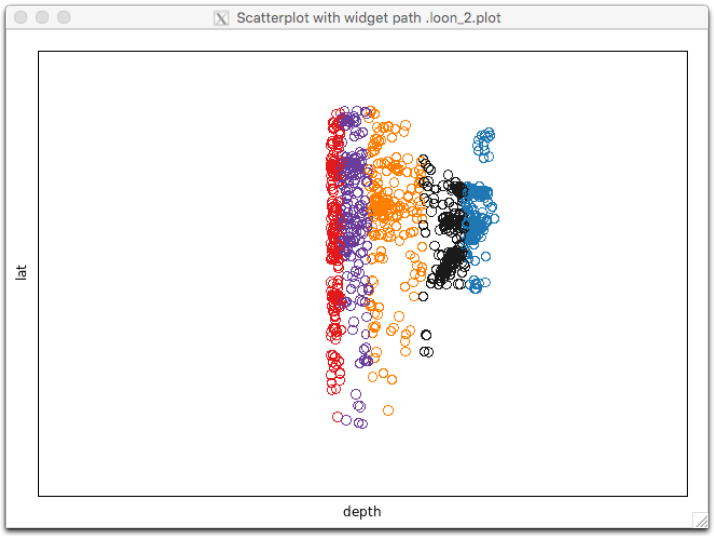




Navigation graphs - navigating along a path

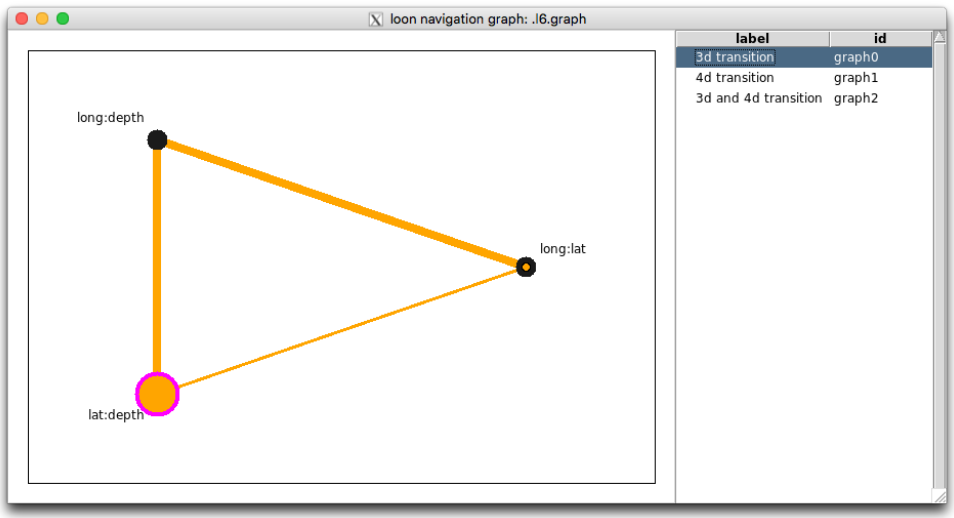
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

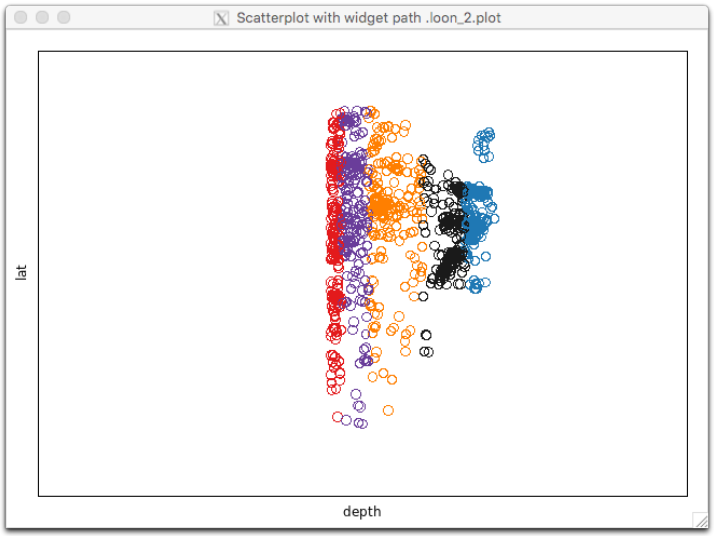




Navigation graphs - navigating along a path

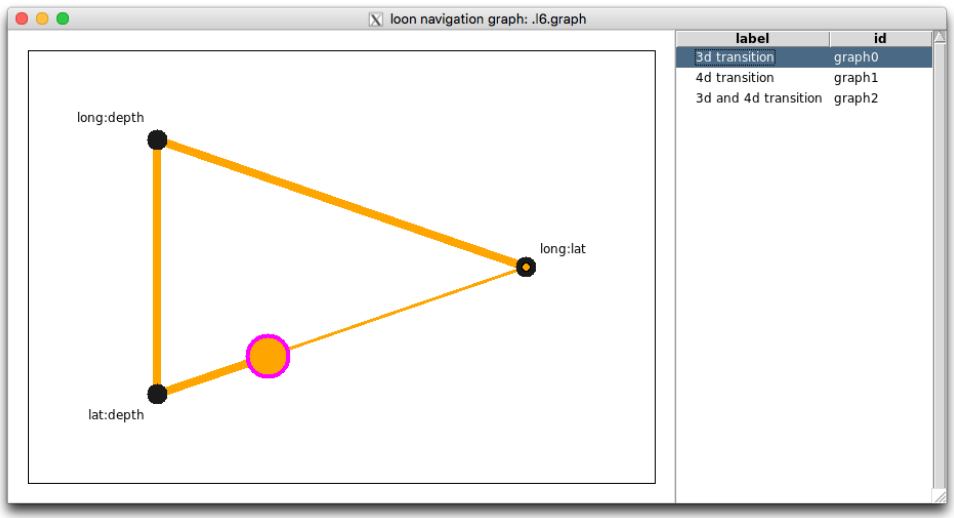
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

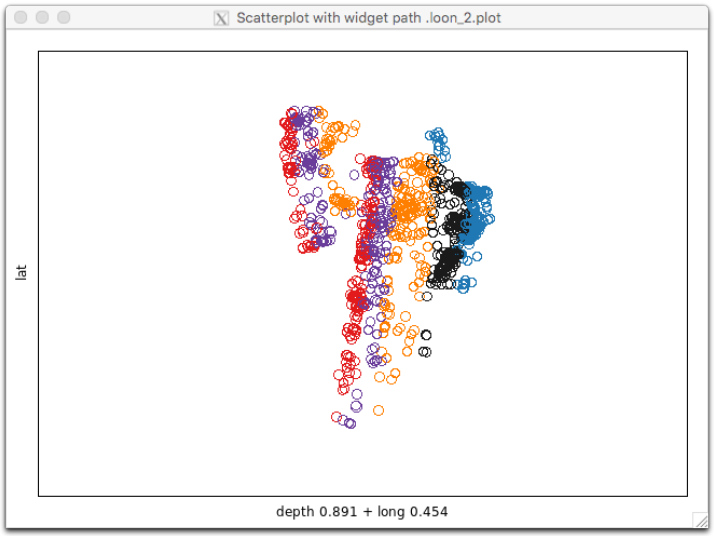




Navigation graphs - navigating along a path

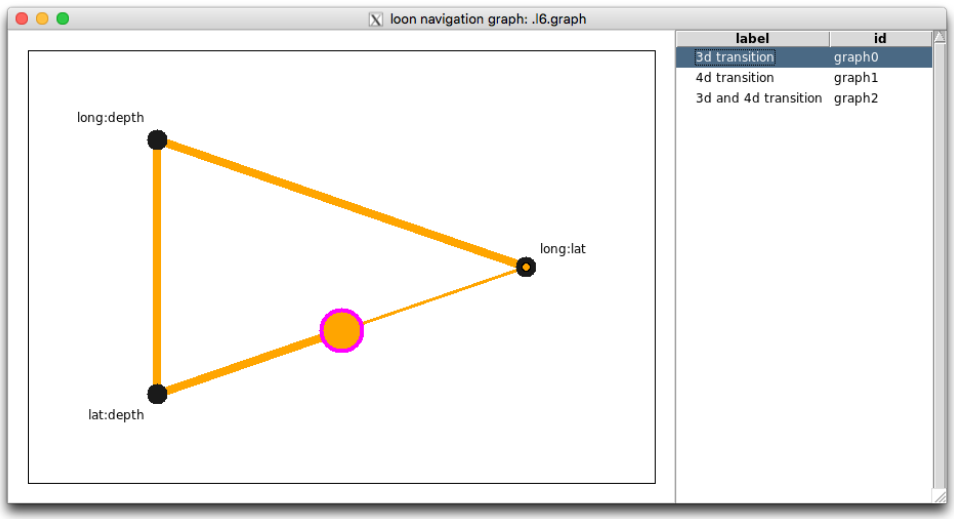
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

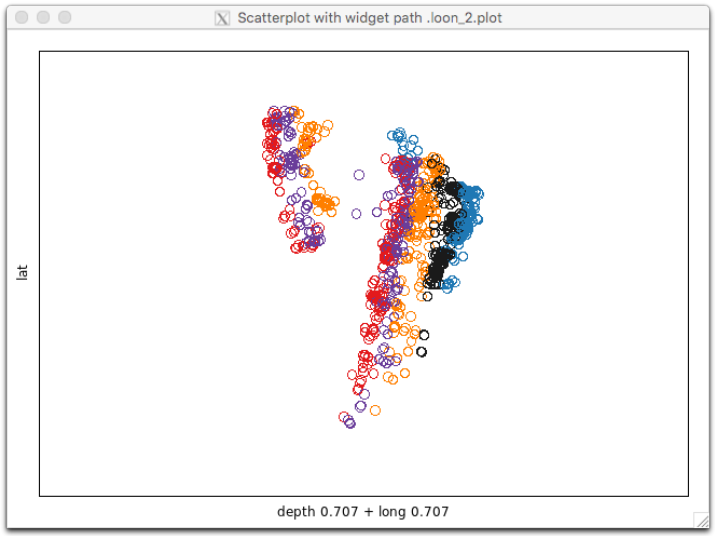




Navigation graphs - navigating along a path

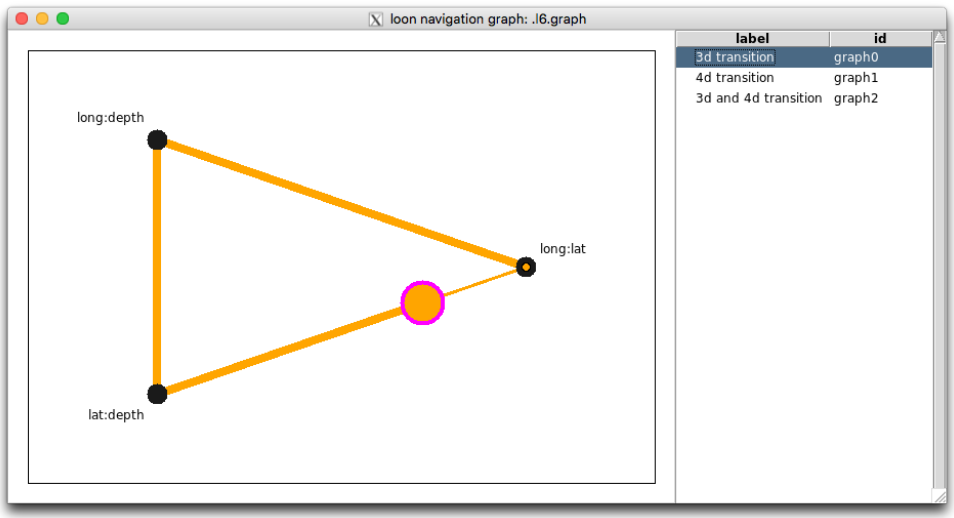
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

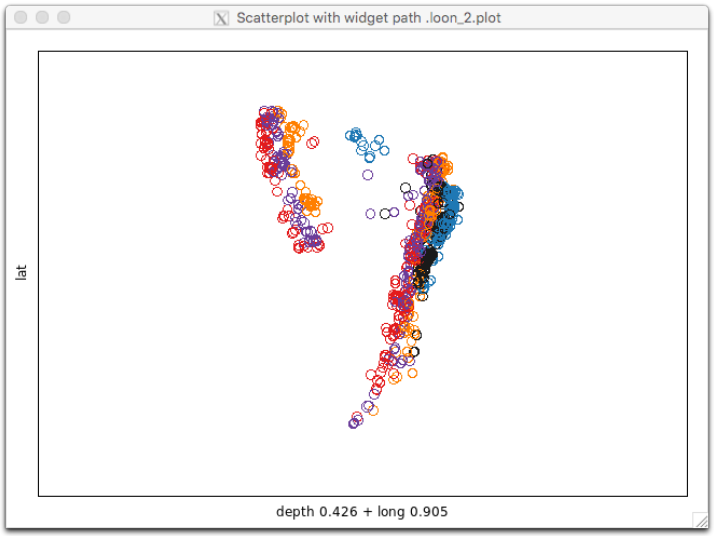




Navigation graphs - navigating along a path

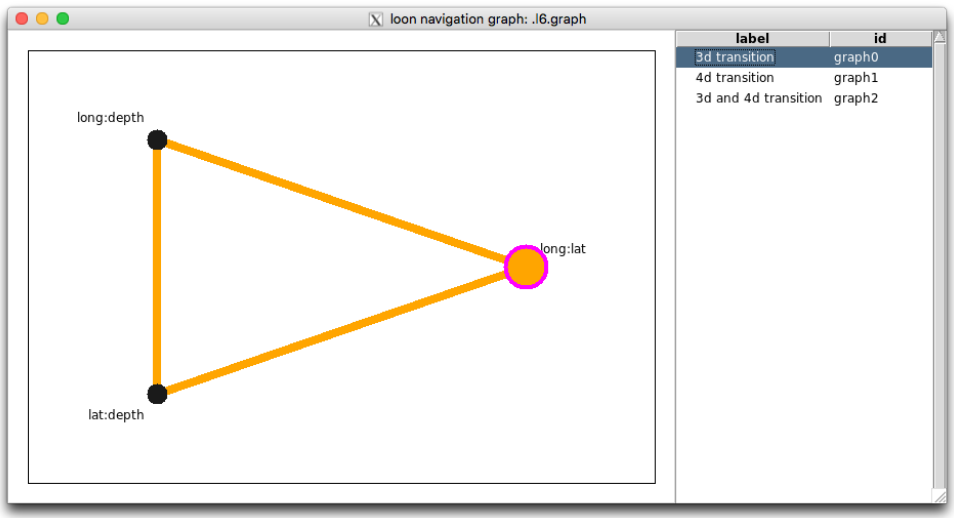
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:





Navigation graphs - navigating along a path

Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:



Navigation graph structure - a `list` of class `l_navgraph`

The navigation graph is a list:

```
str(ng)
```

```
## List of 5
##  $ graph      : 'l_graph' Named chr ".l4.graph"
##  $ plot       : 'l_plot' Named chr ".loon_2.plot"
##  $ graphswitch: 'l_graphswitch' Named chr ".l4.graphswitch"
##  $ navigator  : 'l_navigator' Named chr "navigator0"
##   ..- attr(*, "widget")= chr ".l4.graph"
##  $ context    : 'l_context' Named chr "context0"
##   ..- attr(*, "widget")= chr ".l4.graph"
##   ..- attr(*, "navigator")= chr "navigator0"
##   ..- attr(*, "plot")= 'loon' Named chr ".loon_2.plot"
##  - attr(*, "class")= chr [1:3] "l_navgraph" "l_compound" "loon"
```

Note that the list is of class `l_navgraph` as well as `l_compound` and `loon`. Each element is also a loon object of various classes, including the class `loon`.

For example, programmatic access to the scatterplot can therefore be had via `ng$plot`; similarly for the other elements.

Navigation graph structure - as an `l_compound` object

The navigation graph is an 'l_compound:

Because an `l_navgraph` is also an `l_compound`, the generic function `l_getPlots()` will return all elements which are also loon plots of some kind.

This helper function returns a list of plots with names that should be meaningful for that kind of `l_compound`. For example,

```
l_getPlots(ng)
```

```
## $graph
## [1] ".l4.graph"
## attr(,"class")
## [1] "l_graph" "loon"
##
## $plot
## [1] ".loon_2.plot"
## attr(,"class")
## [1] "l_plot" "loon"
```

Navigation graph structure - the graph and the plot

The graph itself is a special kind of `loon` plot. It has all of the `info_states` of `plot` plus several more related to being a graph. It is in fact a subclass of plot (in `tcl` not in `R`).

For example,

```
g <- ng$graph
names(g)
```

```
##  [1] "itemLabel"        "showItemLabels"   "glyph"            "linkingGroup"
##  [5] "linkingKey"       "zoomX"            "zoomY"            "panX"
##  [9] "panY"             "deltaX"           "deltaY"           "xlabel"
## [13] "ylabel"           "title"            "showLabels"       "showScales"
## [17] "swapAxes"         "showGuides"       "background"       "foreground"
## [21] "guidesBackground" "guidelines"       "minimumMargins"   "labelMargins"
## [25] "scalesMargins"    "x"                "y"                "xTemp"
## [29] "yTemp"            "color"            "selected"         "active"
## [33] "size"             "tag"              "useLoonInspector" "selectBy"
## [37] "selectionLogic"   "activeNavigator"  "nodes"            "from"
## [41] "to"               "isDirected"       "activeEdge"       "colorEdge"
## [45] "orbitDistance"    "orbitAngle"       "showOrbit"
```

```
g["nodes"]
```

```
## [1] "long:lat"   "long:depth" "lat:depth"
```

Generally, programmatic access to the scatterplot via `ng$plot` will be of most interest. Again, both are accessible via `l_getPlots(ng)`

Navigation graph structure - other elements

The remaining components of an `l_navgraph` are generally only of interest in building new displays. These are the `graphswitch`, the `navigator`, and the `context`.

For example, the `navigator` contains information on its display and interaction:

```
nav <- ng$navigator
class(nav)
```

```
## [1] "l_navigator" "loon"
```

```
names(nav)
```

```
## [1] "tag"                          "color"
## [3] "animationPause"               "animationProportionIncrement"
## [5] "scrollProportionIncrement"    "from"
## [7] "to"                           "proportion"
## [9] "label"
```

Case exploration 1: Human immunoglobulin G1 antibody molecule

Here, you will explore the three dimensional structure of the (alpha) carbon atoms making up a single molecule.

```
library(loon.data)
data(igg1)
names(igg1)
```

```
##  [1] "recordType"         "name"               "residue"
##  [4] "chainID"            "residueSequenceNum" "x"
##  [7] "y"                  "z"                  "residueName"
## [10] "group"
```

Case exploration 1: Human immunoglobulin G1 antibody molecule

Here, you will explore the three dimensional structure of the (alpha) carbon atoms making up a single molecule.

```
library(loon.data)
data(igg1)
names(igg1)
```

```
##  [1] "recordType"         "name"               "residue"
##  [4] "chainID"            "residueSequenceNum" "x"
##  [7] "y"                  "z"                  "residueName"
## [10] "group"
```

Now explore the molecule shape as it depends on other variables.

For example, start with

```
igg1_3d <- with(igg1,
                l_plot3D(x, y, z,
                         linkingGroup = "igg1",
                         itemLabel =
                             paste0("Residue: ", residueName, " (",
                                    residue, ")","\n",
                                   "Group: ", group),
                         showItemLabels = TRUE)
                )
```

See what you can learn about this molecule.

Case exploration 2: 1974 Motor Trend Car Road Tests

"The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models)."

```
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

Case exploration 2: 1974 Motor Trend Car Road Tests

"The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models)."

```
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
pp_mtcars1 <- l_pairs(mtcars[, c("mpg", "disp", "hp",
                                 "drat", "wt", "qsec")],
                      linkingGroup = "mtcars",
                      showHistograms = TRUE,
                      itemLabel = row.names(mtcars),
                      showItemLabels = TRUE)

pp_mtcars2 <- l_pairs(mtcars[, c("cyl", "vs", "am",
                                 "gear", "carb")],
                      linkingGroup = "mtcars",
                      showHistograms = TRUE,
                      itemLabel = row.names(mtcars),
                      showItemLabels = TRUE)
fit <- lm(mpg ~ . , data = mtcars)
summary(fit)
```

Case exploration 3: Canadian Visible Minority Data 2006

Population census count of various named visible minority groups in each of 33 major census metropolitan areas of Canada in 2006.

These data are from the 2006 Canadian census, publicly available from Statistics Canada.

Format: A data frame with 33 rows and 18 variates

```
library(loon.data)
data(minority)
names(minority)
```

```
##  [1] "Arab"
##  [2] "Black"
##  [3] "Chinese"
##  [4] "Filipino"
##  [5] "Japanese"
##  [6] "Korean"
##  [7] "Latin.American"
##  [8] "Multiple.visible.minority"
##  [9] "South.Asian"
## [10] "Southeast.Asian"
## [11] "Total.population"
## [12] "Visible.minority.not.included.elsewhere"
## [13] "Visible.minority.population"
## [14] "West.Asian"
## [15] "lat"
## [16] "long"
## [17] "googleLat"
## [18] "googleLong"
```